USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

CacheCatalyst: Enhancing Web Caching for the Latency-Constrained Internet

Mohammad Hosseini, *Shahid Beheshti University;* Sina Darabi, *Università della Svizzera italiana;* Hannaneh B. Pasandi, *University of California, Berkeley;* Patrick Eugster, *Università della Svizzera italiana;* Mahmood Choopani, *Shahid Beheshti University*

https://www.usenix.org/conference/nsdi26/presentation/hosseini

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

CACHECATALYST: Enhancing Web Caching for the Latency-Constrained Internet

Mohammad Hosseini
Shahid Beheshti University

Sina Darabi
Università della Svizzera italiana

Hannaneh B. Pasandi
University of California, Berkeley

Patrick Eugster
Università della Svizzera italiana

Mahmood Choopani
Shahid Beheshti University

Abstract

Caching is a fundamental technique for improving web performance, particularly by reducing page load time through the reuse of previously fetched resources. In this paper, we highlight the drawbacks of the current caching approach, especially in the context of high-speed networks where latency, rather than bandwidth, has become the main bottleneck for web performance. We discuss how the current design of web caching suffers from inefficiencies, particularly due to the latency involved in re-validation requests, which diminishes the potential benefits of caching. To address this inefficiency, we present CACHECATALYST, a novel solution that introduces a early cache validation procedure during the initial step of page loading. This updates the state of cached resources, enabling browsers to utilize unchanged cached content without unnecessary round trips. In addition, our approach enables an optimized form of Server Push that avoids the typical drawbacks of this mechanism. Our evaluations demonstrate that this method improves key web performance metrics by an average of 40%.

1 Introduction

The Web, alongside its underlying protocol HTTP, emerged as the killer application in the early days of the Internet and has continued to dominate the online world. The widespread use of the Web and the dependence of businesses and users on it have made its performance critically important. Numerous industry reports have demonstrated that even small improvements in web performance — such as a 100-millisecond reduction in page load time — can lead to significant gains in user engagement and business revenue [9, 10, 14, 18, 50].

Caching represents one of the most effective strategies for improving web performance by enabling client browsers to reuse previously fetched resources. When a browser loads a web page, it stores reusable resources, which can then be utilized in subsequent requests to the same page or other pages within the same website. This eliminates the time spent downloading these resources, thereby significantly reducing page load time.

Despite its benefits, the current web caching mechanism suffers from significant limitations that prevent it from reaching its full potential. Design flaws in the existing caching system lead web developers to use it conservatively [26, 27, 41]. This means that many resources are either not made cacheable or have their cache duration set to a short period. This conservative approach, at best, leads to the underutilization of caching benefits and, at worst, results in frequent unnecessary data transmissions [25, 34, 40]. Hence, there is a significant gap between the ideal and the achieved performance of web caching, highlighting the suboptimality of current caching approach [26, 40].

Apart from the conservative use of caching, another issue is that the HTTP cache re-validation mechanism, which is used to avoid retransmitting unchanged content, has inherent inefficiencies. In this mechanism, the client sends a request for a resource along with the validation token (*ETag*) of the resource stored in its cache. The server compares this token to the current version and, if the resource is unchanged, returns a short response indicating that no download is needed. However, this solution is rooted in the limitations of the early Internet era, when bandwidth was low, and download speed was the primary bottleneck. Nowadays, with the high download speeds of the core and access links of the Internet, the primary bottleneck has shifted to latency. Re-validation requests still necessitate a round-trip time (RTT), which can be significant and sometimes even longer than the transmission time for downloading a resource.

This issue becomes highly significant considering that, with today's high-throughput access links, end-to-end latency plays a crucial role in page load time. Early research by Sundaresan *et al.* revealed that at a downstream throughput of 16 Mbits/s, the latency of the access link becomes the main bottleneck for web page load time [51]. They have shown that even a 10-millisecond increase in last-mile latency can lead to an increase of several hundred milliseconds in page load time. This issue is even more pronounced with mobile web access.

Cellular networks, while offering high download speeds, also have considerably high latency; even 5G links often exhibit latency similar to that of 4G [38]. Recent research on mobile web performance shows that latency, rather than bandwidth, is the primary factor influencing page load time [28, 32, 33, 41]. Therefore, minimizing the number of RTTs required to load a web page is essential.

Two well-known approaches for addressing this issue are HTTP/2 Server Push and Remote Dependency Resolution (RDR). Using the Server Push mechanism, the server proactively transmits resources needed for loading the requested page, without considering which resources are already present in the client's cache. Due to significant performance issues with Server Push—such as increased data usage costs, higher bandwidth overhead on servers, and longer page load time caused by computational overhead on the browser [61, 62]—Chrome discontinued support for the feature in version 106 (released in 2022) [12], followed by Firefox, which removed it in version 132 (released in 2024) [30]. RDR-based solutions employ a proxy server that has a lower RTT to the web server than the RTT between the client and the web server. The proxy retrieves the complete page from the web server and then delivers all resources to the client in one batch, consuming a single proxy-client RTT. However, this approach poses security and privacy issues because of the proxy acting as a man-in-the-middle for TLS connections.

An effective way to reduce the latency impact on page load time is to eliminate the RTTs caused by unnecessary cache re-validation requests, and it requires a rethinking of the web caching mechanism. To address the inefficiency of the current caching design, we propose **CACHECATALYST**, a novel approach that eliminates unnecessary round trips caused by the cache re-validation mechanism. The key idea of our approach is to establish an efficient early cache validation mechanism during the initial stage of web page loading. This mechanism enables two crucial determinations: it allows the client to identify which of its cached resources remain valid, while simultaneously enabling the server to determine which required resources the client lacks in their up-to-date versions.

CACHECATALYST introduces a modified request–response pattern: when requesting the base HTML file, the client includes metadata about its cached resources, specifically the URLs and their corresponding validation tokens. The server includes, within its response to the base HTML file request, information about which of the client's cached resources are stale and which remain valid. This early exchange of validation information serves two purposes; First, it acts as a catalyst for the browser cache, enabling the browser to immediately utilize valid cached resources after receiving the base HTML file, without incurring additional RTTs. Second, it allows the server to optimally push only those required resources that are either missing from the client's cache or present in outdated versions, thereby avoiding redundant transmissions of resources already present in the browser's cache. We refer to this type of Server Push as *cache-aware push*. The proposed solution significantly accelerates the page loading process by eliminating unnecessary requests for content that remains unaltered. Moreover, this approach eliminates the need to set cache duration (`max-age`) for resources, and thus resolves the issue of conservative cache usage. Our evaluations show that CACHECATALYST improves not only Page Load Time (PLT), but also user-centric web performance metrics—including First Contentful Paint (FCP), Largest Contentful Paint (LCP), Speed Index (SI), and Time to Interactive (TTI)—by an average of 40%, thereby significantly enhancing the web user experience through improved perceived visual completeness, smoothness, and interactivity.

Our contributions can be summarized as follows:

- Through empirical analysis, we show that the current web caching mechanism is inadequate for today's Internet, where latency—rather than download bandwidth—has become the primary bottleneck (§3).
- We introduce CACHECATALYST (§4), a novel caching scheme that eliminates most unnecessary RTTs caused by cache re-validation requests, thereby significantly reducing PLT, FCP, LCP, SI and TTI, and ultimately enhancing the overall web user experience (§6). We have made the source code of our implementation (§5) publicly available.[1]
- Our method eliminates the need to set speculative cache durations (`max-age`) for web resources. This not only simplifies web page design but also resolves the issue of conservative cache usage.
- Our approach enables optimal and seamless use of Server Push during page revisits, without its well-known drawbacks. We introduce a cache-aware push mechanism and demonstrate its benefits through both implementation and evaluation, comparing it against HTTP/2 Server Push. While standard Server Push often incurs heavy traffic overhead, reduces server responsiveness, and degrades user experience, our cache-aware push avoids these issues and instead improves user-centric performance metrics.
- We have designed and implemented this new caching approach to be fully compatible with existing browsers, allowing for seamless deployment without requiring any modifications to client-side software (§5).[2]

2 Background and Motivation

Before exploring the motivations for rethinking web caching, we briefly review the current web caching mechanism and its deployment through HTTP cache headers.

[1] https://github.com/toorin-lab/CacheCatalyst

[2] This paper is an extended version of our recent workshop publication [17]

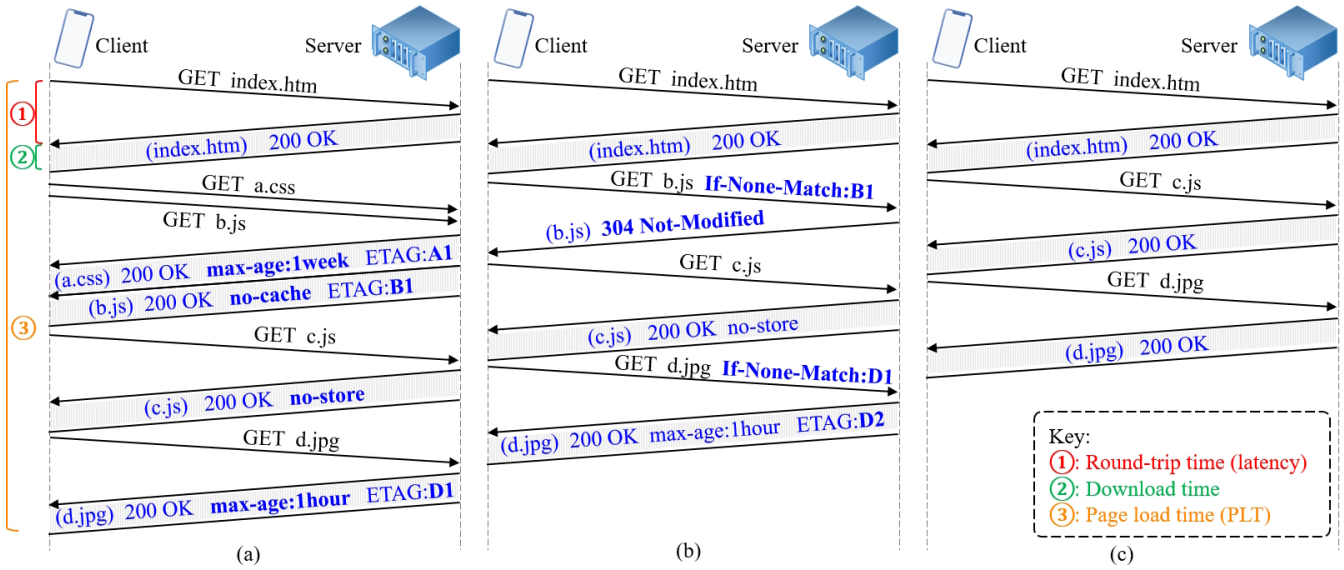


Figure 1: Web caching mechanism using HTTP cache headers
(a) The first visit to a web page. "`a.css`" and "`b.js`" are identified after parsing "`index.html`". "`c.js`" and "`d.jpg`" are fetched as a result of evaluating "`b.js`" and "`c.js`", respectively. **(b)** Another visit to the same page or a similar page on the same website two hours later. **(c)** Optimized scenario for a revisit, assuming "`a.css`", and "`b.js`", and "`d.jpg`" have not changed at the time of the revisit.

2.1 Overview of the current web caching

Each web page is composed of a base HTML file and numerous other resources linked within this HTML file. A client browser retrieves the web page and its resources from a server using the HTTP protocol. Figure 1a shows an example of HTTP interactions to retrieve a web page. The browser first sends an HTTP `GET` request to the server to fetch the base HTML file (`index.htm` in the figure). By parsing the HTML file, the browser identifies links to other resources (`a.css` and `b.js`) and sends requests for them as well. Fetching and evaluating other resources (such as CSS and JavaScript files) may lead to the need for additional resources (`c.js` and `d.jpg`), in which case the browser fetches them too. As illustrated in the figure, the time required to fetch each resource consists of two components: round-trip time (RTT) and the transmission time of downloading the resource. The RTT is primarily determined by the distance between the client and the server, while the download time depends on both the bandwidth of the Internet links connecting them and the size of the resource.

Since many resources may be requested again during subsequent visits to the same page or other pages on the same website [22], the browser caches these resources based on cache-control directives sent by the server in the header of each response. The most important cache directives are `no-store`, `no-cache`, `max-age`, and `Etag`. These directives inform the browser how long to keep each resource in the cache and what actions to take when the resource expires. When the browser needs a resource again, it first checks its cache to see if a copy is already available. If the resource is found in the cache and has not expired, the browser quickly serves the cached content. If the resource has expired or if the browser was instructed not to cache it, the browser requests a fresh copy from the server.

The simplest cache directive is `no-store`. This directive instructs the browser not to cache the resource, requiring it to download the content every time it is needed. It is typically used for sensitive or frequently updated dynamic content that should never be stored locally. The `max-age` directive specifies the time-to-live (TTL) for a resource, indicating how long its content will remain unchanged. The browser can cache and reuse a resource for the duration specified by its `max-age` field. Figure 1b shows a revisit to the mentioned page two hours after the first visit. Since `a.css` was retrieved with a `max-age` of one week in the first visit, the browser does not send a request for this file in the second visit and instead uses the cached content. If a resource is expired (`d.jpg` in the figure), the browser sends a conditional request known as the re-validation mechanism. The `no-cache` header (`b.js` in the figure) results in the same outcome. Despite its name, `no-cache` means that the browser can cache the resource, but it must re-validate it before each reuse. A re-validation request asks the server to send the resource if its content differs from the cached version in the browser. If the resource has not changed, the server sends a short `Not-Modified` response, thereby eliminating the transmission time of (re-)downloading the resource. This is the case for `b.js` in Figure 1b. If the resource has changed, the server sends the new version of the

resource, which is the case for `d.jpg` in Figure 1b. Checking for changes is done using a validation token called `Etag`. The server includes a validation token in each response that can be cached. The browser includes the validation token in the `If-None-Match` field of each re-validation request to inform the server which version of the resource is cached in the browser.

2.2 The need for rethinking web caching

The current web caching mechanism is suboptimal and requires rethinking. In the following, we examine the reasons necessitating this rethinking.

The latency bottleneck. A seminal study in 2013 revealed a significant finding: increasing bandwidth beyond a certain threshold[3] does not substantially reduce web page load times [51]. At this throughput rate, network latency emerges as the primary bottleneck and the determining factor in PLT. The research demonstrated that despite higher average throughput in East Asia compared to Europe, users in the former region do not experience proportional improvements in PLT due to latency bottlenecks negating the benefits of increased throughput. Subsequent research by Asrese *et al.* [3] corroborated these findings, confirming that webpage performance varies across regions and service providers without a direct correlation to throughput.

Latency is primarily determined by the geographical distance between client and server, with server processing time playing a secondary role. Geographically distributed CDN servers help alleviate the problem. Nevertheless, the impracticality of placing servers close to all users, together with the inherently high latency of certain access links (e.g., mobile), poses significant challenges to latency reduction. Consequently, improving page load time requires minimizing the negative effects of latency. Caching is among the most effective approaches. However, as will be discussed below, the current caching mechanism is designed to address the limitations of low throughput, not the challenges of high latency.

Significant redundant transfers in web traffic. Basically, cache directives for resources should be set by the website developer, who can determine whether a resource should be cached and for how long (TTL) based on its type, usage, and content. However, since this task is difficult, developers often neglect it. In such cases, the website's content management system (CMS) applies default directives according to resource type. Due to the CMS's lack of specific knowledge about each resource's usage, it frequently assigns settings that prevent client-side caching for many resources. Consequently, many resources that could be cached are not cached in practice, and the same content is repeatedly downloaded on each visit. Empirical studies have demonstrated that, on average, only approximately 50 percent of cacheable resources are effectively cached [25–27]. The prevalence of redundant transmissions in web traffic further corroborates this issue [1,25,34,40]. These redundant transmissions not only increase PLT but also result in higher power consumption and data costs for mobile users. Hence, we need a more straightforward caching solution that can be optimally deployed without requiring direct developer intervention.

Conservative TTLs. In the current caching approach, web servers are required to assign a Time To Live (TTL) value for each cacheable resource[4]. However, accurately estimating how long a resource will remain unchanged is challenging, and predicting the exact moment of modification is usually impossible. Setting a TTL that is too long can cause the client to use a stale version of the resource. Conversely, setting a TTL that is too short causes the resource to expire prematurely, leading to unnecessary re-validation requests to the server even though the content has not changed. These re-validation requests impose additional RTTs on the page load time, which make the current caching scheme particularly ineffective for mobile web browsing [58,59]. Empirical studies have shown that developers tend to set TTLs conservatively, often significantly shorter than the actual duration of content stability. For instance, the research by Ramanujam *et al.* has shown that 47% of resources expire in the cache even though their content has not changed [41]. Similarly, Liu *et al.* found that while 40% of resources have a TTL of less than one day, 86% of these do not change within that period, resulting in unnecessary cache expiration [26]. The most challenging scenario arises when TTL estimation for a resource is not possible. In such cases, developers resort to using the `no-cache` directive. This approach, while allowing the resource to be cached, designates it as perpetually stale, compelling the client to re-validate it upon each use.

The aforementioned caching policy was beneficial when the Internet bandwidth of clients was low and downloading was a bottleneck, as it improved page load time by eliminating the transmission time of downloading unchanged resources. However, in the context of contemporary high-throughput connections, this approach proves insufficient, particularly considering the relatively small size of resources on modern web pages. According to the 2025 web resource statistics [21], the average size of each resource type is 10 KB for HTML, 10 KB for CSS, 30 KB for JavaScript, 30 KB for fonts, and 50 KB for images. Using a modest 20 Mbps connection, most of these resource types can be downloaded within 4 to 20 ms. Meanwhile, the average RTT is around 50 ms for intercity communications, and typically ranges from 100 to 200 ms for transcontinental and transoceanic communications [29]. It should be noted that CDN servers do not fully resolve this issue. Deploying CDN servers in every city worldwide is impractical, and many websites cannot afford the associated costs. Furthermore, CDNs cannot address the latency that inherently arises from access links. For example, the median

[3] 16 Mbps at that time.

[4] Using the `max-age` directive.

latency of global 5G Internet access is about 40 ms, which corresponds to an RTT of roughly 80 ms [49]. Given these circumstances, merely eliminating download time is no longer adequate; it is now crucial to also mitigate or eliminate re-validation RTTs. Therefore, an enhanced caching solution is required—one that not only avoids reliance on TTL values but also eliminates re-validation requests. An ideal caching solution should ensure that a client requests a new version of a resource only when the content of the cached resource has actually changed on the server. In essence, as long as the resource remains unmodified, the client should be able to utilize it immediately without incurring the cost of an RTT. Figure 1c illustrates this optimal scenario.

2.3 Server Push

A well-known approach for reducing RTTs is the Server Push mechanism, which is a feature of HTTP/2 [4]. This mechanism enables the server to proactively transmit resources necessary for rendering a web page to the client browser prior to explicit requests from the client. The server can transmit these resources concurrently with the base HTML file response. Ideally, this allows the client browser to receive the latest version of all resources with just one RTT. However, in practice, the number of RTTs does not reduce to one due to the inability to predict all the resources the user will need and the impossibility of pushing cross-origin resources hosted on third-party domains.

Server Push sends resources without considering those already present in the browser's cache, leading to wasted user bandwidth and increased data usage costs [55, 60]. While this issue might not be significant for users with high bandwidth availability, it creates severe limitations for web servers [19, 20]. A server simultaneously handles requests from a large number of clients, and if it sends all resources regardless of the cached ones on the client side, the number of users it can serve concurrently is significantly reduced. Additionally, the extra load imposed on the browser by delivering these resources degrades page load time and user-centric performance metrics [16, 61, 62]. These issues ultimately led major browsers like Chrome and Firefox to discontinue support for this feature [12, 30].

3 Empirical Analysis

To assess the potential impact of eliminating unnecessary RTTs on page load times, we need to examine the frequency of `Not-Modified` responses and their timing characteristics. For this purpose, we conducted an experiment on 1000 websites using Chrome's Lighthouse tool [13]. These 1000 websites were randomly chosen from the top 100K most visited websites [39, 52], with one sample taken from each consecutive rank group of 100 (1–100, 101–200, and so on). Since the homepage of a website might not be a good representative of

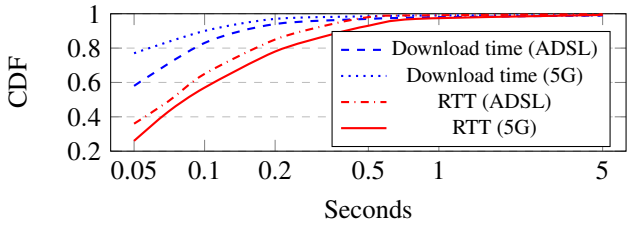


Figure 2: Download time and RTT for resources of the selected web pages, truncated at P99 and P99.7, respectively.

the website's structure [2], we randomly selected 9 additional internal pages from each website by examining its homepage source, bringing our total sample to 10,000 web pages. The experiment was performed using two distinct Internet access links: an ADSL link and a 5G cellular connection, with download bandwidths of about 20 and 40 Mbps, respectively. We initially loaded each website to warm up the browser cache. Lighthouse provides detailed timing information for each resource, including connection establishment, server response time, and download time. We denote the sum of connection establishment time and server response time as RTT. Figure 2 shows the distributions of RTTs and download times measured across website resources. The results show that, for the majority of resources, RTTs are not only on the same order as download times but often exceed them. For example, under 5G, the median download time was about 35 ms, whereas the median RTT was roughly 90 ms. This observation underscores the inadequacy of focusing solely on eliminating download time while disregarding RTT in efforts to optimize page load performance. It is worth noting that the download bandwidth available to users in many countries is significantly higher than in our experiments [37], which makes the impact of RTT on page load performance more pronounced.

Following the initial page load, we reloaded each page at intervals ranging from one minute to one hour and counted the number of conditional requests and `Not-Modified` responses. Figure 3 illustrates the proportion of requests that were conditional and the proportion of responses that returned a `Not-Modified` status. This visualization provides insight into the frequency of cache validation attempts and their outcomes. As observed, over 75% of the requests are conditional, and more than 65% of all responses are `Not-Modified`. As the time interval between reloads increases, the percentage of conditional requests rises due to the expiration of some resources. Interestingly, the relative difference between the number of conditional requests and `Not-Modified` responses decreases as the reload interval increases, indicating that most expired resources had not actually changed.

The results suggest that eliminating unnecessary requests could significantly reduce page load time. However, because browsers typically fetch independent resources concurrently, the actual benefit depends on how much the critical path is

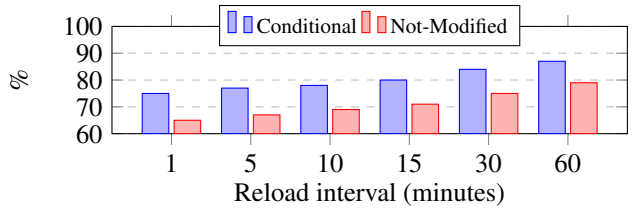


Figure 3: Measuring conditional requests and `Not-Modified` responses across reload intervals.

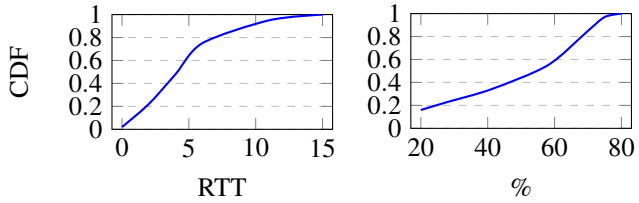


Figure 4: Reduction in critical path length by eliminating conditional requests for unchanged resources.

shortened. The critical path is the longest chain of dependent resources that the browser must download and process to render a webpage. Utilizing the Lighthouse tool, we measured the critical path length for the web pages after reloading, quantified in terms of the number of resources, which is equivalent to the number of RTTs. The critical path length varied between 2 and 19 resources across the tested pages, with an average of 6.7. Subsequently, we eliminated the resources that had received a `Not-Modified` response and remeasured the critical path. Sometimes the original critical path becomes shorter, while in other cases a different path emerges as the critical path. Figure 4 shows the reduction in the critical path length, which proves to be substantial. It is expected that by removing the unnecessary conditional requests, page load times could potentially be reduced by an average of 56%. However, not all web performance metrics have a direct one-to-one relationship with the critical path length. Hence, our evaluations in Section 6 show that the proposed approach, which eliminates unnecessary requests, yields improvements smaller than this estimate, though still substantial.

4 CacheCatalyst

In this section, we propose CacheCatalyst, a simple solution for eliminating the RTTs imposed by unnecessary cache re-validation requests. We begin with an overview of the solution and then describe its components in detail.

4.1 Overview

In the CacheCatalyst solution, the browser and web server perform an early cache validation during the initial stage of web page loading. After the first stage of loading a web page—sending a request for the base HTML file from the client browser and receiving the response from the web server—the browser updates the state of its cached resources. The early cache validation performed during this initial stage allows the browser to identify which of its cached resources are up-to-date and usable, and which ones are stale and need to be re-fetched. The browser then parses the base HTML file and identifies links to additional resources. For each resource, it queries its local cache. If a cached resource is stale, this indicates it has been modified on the server and must be re-fetched. Conversely, if the cached resource is up-to-date, the browser utilizes the cached content. This approach eliminates the need for re-validation requests, allowing the browser to reuse the cached content of unchanged resources without incurring additional RTTs. Furthermore, it removes the need to specify TTL values or set `max-age` parameters, thereby simplifying the caching process. Notably, this mechanism is not restricted to the initial page load; it also applies to subsequent asynchronous Ajax requests.

To implement this solution, three primary challenges must be addressed. The first challenge is how to perform early cache validation, which involves updating the state of browser-cached resources associated with the requested webpage. The second challenge stems from cross-origin resources. Certain elements of a web page may originate from hostnames distinct from that of the base HTML file, indicating their hosting on third-party servers. Consequently, the primary server of the web page lacks control over these resources and cannot determine their status. The third challenge lies in implementing this caching model on the client side, specifically in adapting browsers to prioritize the utilization of the proposed caching method over traditional approaches. In the following, we discuss in turn how we address these challenges.

4.2 Early cache validation

CacheCatalyst performs early cache validation by leveraging the client browser's existing knowledge of previously fetched resources. Rather than requiring the server to predict or enumerate all resources needed for a page, the browser proactively informs the server about relevant cached resources when initiating a page load. This design choice is essential for supporting modern web pages that include dynamic, user-dependent, and JavaScript-generated resources.

When requesting the base HTML file, the browser includes metadata about its cached resources for the target domain—specifically, the URLs of cached objects and their associated validation tokens (`ETags`)—in a dedicated request header. Upon receiving this request, the server compares the provided validation tokens against its current versions and returns, in the response headers, updated validation information indicating which cached resources remain valid and which have become stale. Using this information, the browser up-

dates the state of its cache immediately after receiving the base HTML file, before requesting any dependent resources.

A key challenge in this design is that the browser might have cached a large number of resources for the target website. For example, this is often the case with e-commerce websites. By navigating through different pages of a store and viewing various products, a significant number of resources are stored in the browser's cache. Therefore, it is not practical to simply send URLs of all cached resources related to the domain of the target page to the server. This would not only increase the size of the request but also impose a considerable overhead on both the browser and the server.

To mitigate this, the browser can store, for each cached resource, the pages on which it has been used. When revisiting a page, the browser can then determine which of the cached resources are relevant to that page and only include URLs of those resources in the request. This approach, however, limits early cache validation to revisits of previously visited pages. If the user loads a new page of a previously visited website, no information about the domain's cached resources is sent to the server—even though many of the required resources may already reside in the browser's cache, as resources are often shared across pages of a website [22]. To address this, we propose that the browser include the most frequently used cached resources for that domain, as well as those that were recently used, in the early cache validation process. The most frequently used resources are likely to be shared across multiple pages of the website, while recently used resources may include items the user is likely to request again (e.g., a product added to the shopping cart). This enhancement updates the state of many relevant cached resources for a new page before they are requested. Notably, resources that remain unvalidated during this process can still be retrieved from the cache via conditional requests, though subject to the additional RTT inherent in the conventional HTTP caching mechanism.

4.3 Cross-origin resources

A web page may include cross-origin resources, which are resources hosted on servers different from the one delivering the requested page. The hostname in the URL of these resources differs from the web server's hostname. The challenge these resources pose in our proposed solution is that the web server cannot verify their validation tokens. To address this challenge, CACHECATALYST has the browser verify the validation tokens of cross-origin resources with their respective hosting servers concurrently with requesting the base HTML file. The browser can use the HTTP `HEAD` method to verify these validation tokens. The `HEAD` method requests only the response headers from the host server, omitting the resource content. A conditional `GET` request can also be used for this purpose so that the server includes the content in its response if the resource has been modified. However, since many cross-origin resources are not part of the primary content of the pages (e.g., advertisements) [5, 22, 24], it is not recommended to send all of them to the browser during the initial page load, as this would impose unnecessary overhead on the browser [43]. Hence, we categorize cross-origin resources into two groups: those from a subdomain of the page's main domain, likely primary resources, and those from a third-party domain, likely not part of the primary content. For the first group, we send a conditional `GET` request, and for the second group, we send a `HEAD` request.

A potential limitation of our approach arises from Cross-Origin Resource Sharing (CORS), a browser-enforced security mechanism that restricts how resources hosted on one origin can be accessed by documents or scripts from another. CORS restrictions apply only to cross-origin resources fetched programmatically via JavaScript (e.g., using `fetch` or `XMLHttpRequest`) [31]. Resources embedded directly in HTML—such as images, stylesheets, fonts, icons, and many advertisements—are not subject to these restrictions and can be loaded cross-origin without limitation. Since a large fraction of cross-origin resources fall into this category, CORS does not fundamentally limit CACHECATALYST's effectiveness. We evaluate the impact of CORS-related constraints in more detail in Section 6.

4.4 Cache-aware push

As previously mentioned, HTTP/2 Server Push has been deprecated due to its drawbacks, with browsers like Chrome and Firefox no longer supporting this feature [12, 30]. The main reason for the inefficiency of Server Push is that it ignores the browser cache. The server blindly pushes resources without considering their presence in the browser cache, resulting in unnecessary server load, browser overhead, bandwidth waste, and energy consumption. Although HTTP/2 provides a mechanism for the browser to stop the server from sending unnecessary resources by issuing an `RST_STREAM` frame with a `CANCEL`/`REFUSED_STREAM` code [4], this solution has not been successful in practice. The likely reason is the high bandwidth-delay product: by the time the browser detects that the server is sending a resource it does not need, the server may have already sent the resource in full, and the data packets are already in transit. Furthermore, another limitation of Server Push is that the server cannot predict and push user-specific dynamic resources.

We do not aim to fully revive Server Push, but our proposed solution enables optimized and problem-free use of Server Push during revisits to a web page. In addition to URLs, the browser can include the validation tokens of the cached resources in the request for the base HTML file. Therefore, the server identifies at the beginning of the page loading process which up-to-date resources are available in the browser's cache. This allows the server to push only those resources that are not up-to-date or missing from the browser cache. As a result, no unnecessary load is placed on either the server or the

browser, and bandwidth waste is effectively prevented. Moreover, this approach allows the server to identify the dynamic resources required by the user and push their updated versions if needed. Therefore, we recommend using this cache-aware server push in conjunction with our proposed solution during revisits to a web page. In our implementation, which is described in the next section, we have designed this method to work seamlessly even with existing browsers that have removed support for Server Push.

5 Implementation

Although server-side implementation of CACHECATALYST is relatively straightforward, the key issue is how to apply it to current browsers so that users of existing browsers can benefit from this solution.

5.1 Client-side adaption

An important challenge is to modify the client-side browser's caching mechanism to work with the proposed approach. The browser needs to include the validation tokens (`Etags`) of the cached resources in the request for the base HTML file. Based on the server's response, the browser must update the state of the resources in the cache by removing those that have become stale and keeping the rest for potential use on that page. Ideally, this method would be intrinsically integrated into the browser. In the absence of such native support, we propose an alternative solution to implement the CacheCatalyst approach using existing browsers without requiring any modifications to them. This solution leverages the Service Worker API [53].

A Service Worker, as illustrated in Figure 5, functions as a proxy that resides within browsers, intercepting and managing requests and responses between a website's client-side interface and its origin server. Service Workers are domain-specific, meaning that each Service Worker is registered by the origin server of a particular website and operates exclusively within the scope of its corresponding domain. A Service Worker, which is written in JavaScript, can control, modify, and even respond to requests independently. One of the key features of Service Workers is their ability to maintain their own cache, which can be leveraged to respond to requests even when the origin server is inaccessible, such as in offline mode. All major desktop and mobile browsers (Chrome, Firefox, Safari, and Edge) support Service Workers.

In the CACHECATALYST solution, the web server of a website registers a Service Worker associated with the website's domain on the browser. This registration is carried out during the client's first visit to the website. The Service Worker stores all resources received from the server in its cache, provided they do not have a `no-store` header. It also keeps track of the resources fetched during the last visit to each page. During revisits to any page, when the Service Worker intercepts the request for the base HTML file, it adds information about

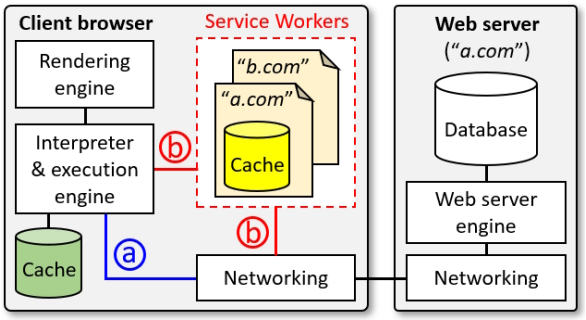


Figure 5: Service Workers.
ⓐ: The path that requests and responses normally take
ⓑ: The path taken if a Service Worker exists for a domain

the cached resources for that page to a header field named `X-Cached-Resources` and then forwards the request to the server. If the request is for a page that has not been visited before, the Service Worker adds information about the most frequently used and recently used cached resources for that domain to the request. In our implementation, this selection consists of the 50 most frequently used resources and the 50 most recently used resources. Upon receiving the server's response, which includes the updated validation tokens for those resources, the Service Worker updates the status of those resources in the cache and removes the ones that have become stale. Hence, after receiving the response for the base HTML file, the validation tokens in the Service Worker's cache table are up-to-date. Consequently, conditional requests whose validation tokens match those in the Service Worker's cache table can be immediately served by the Service Worker. Despite all this, the Service Worker may still receive requests for resources that are not present in its cache. For example, it could be a new resource of the requested page or a dynamic resource that had not been fetched during the previous visit to that page. Additionally, other cached resources whose information was not included in the browser's initial request remain unvalidated, and their freshness status is unknown. If a request for them is made, the Service Worker cannot serve them from its cache. In all these cases, the Service Worker forwards the conditional request to the web server.

The cache-aware push can also be implemented using the Service Worker. The CACHECATALYST Service Worker, alongside the request for the base HTML file, also sends another request called `Push_Request`. If this feature is enabled, the web server responds to this request by sending the content of resources that are outdated on the client side, allowing these resources to be updated in the Service Worker cache.

5.2 Server-side implementation

We implemented the CACHECATALYST solution using Caddy [7, 8], a popular open-source web server. We modified Caddy so that when it receives a request for a base

HTML file, it identifies the resources specified in the `X-Cached-Resources` header and includes their updated `Etags` in a header named `X-Etag-Values` in the response sent to the client. Additionally, the web server integrates the Service Worker registration code into the base HTML file. Another task that our modified Caddy performs is sending the resources that are not up-to-date in the client's cache in response to a `Push_Request`. This is done using the same simple HTTP request and response mechanism, and the Service Worker, upon receiving the response, stores the content of the resources in its cache. Therefore, our proposed cache-aware push mechanism also works on current browsers.

6 Evaluation

To evaluate CACHECATALYST, we must examine its impact on performance metrics when revisiting a page or navigating to another page from a previously visited website. To ensure the results are as realistic as possible, we rely on real-world websites. However, since this approach requires modifications on the web server side, we cannot perform the evaluation directly on live websites. Therefore, we created a controlled test network to host cloned versions of websites and conduct our experiments on them. The testbed comprises three virtual machines—a client, a main web server, and a third-party web server—all interconnected via an Open vSwitch in a star topology. The client requests a page hosted on the main server, while the third-party server provides cross-origin resources not belonging to the page's domain or its subdomains. Another advantage of this test environment is that it allows us to emulate diverse network conditions; for this purpose, we used the Linux `tc` (Traffic Control) utility to configure the bandwidth and latency of the links.

We cloned the 10,000 pages described in Section 3. For each page, we stored six versions: the original copy and five additional versions captured at intervals of one minute, one hour, six hours, one day, and one week after the first. In our experiments, before each revisit, we replace the original version on the main web server with the corresponding copy to reflect the intended page changes. To accurately simulate the third-party web server scenario, we preserved the original structure of resource links within the files of each web page. If we had converted all links to local links, it would have been impossible to identify cross-origin resources. Furthermore, links generated through JavaScript execution may not be local, and we needed to handle these as well. For each web page, we stored all resources within its domain on the main web server, while cross-origin resources from other domains were stored on the third-party web server. We also captured the cache control header of each resource. To ensure proper routing of client requests for these resources to the appropriate main or third-party web server, we configured DNS entries for the relevant domains within the DNS cache table of the client machine. We equipped the web servers with the modified version

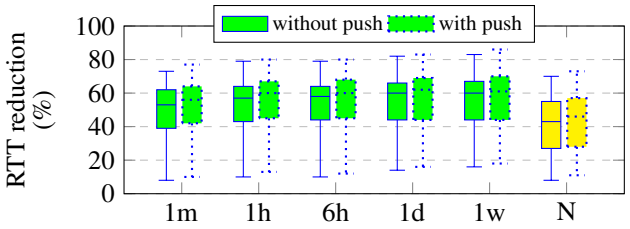


Figure 6: Reduction in the number of RTTs achieved by CACHECATALYST in two scenarios: first, reloading the web pages (labels 1m to 1w, which indicate reload delay); second, visiting new pages on a previously visited website (label N)

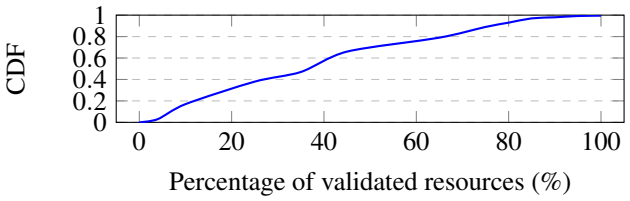


Figure 7: The coverage of the early cache validation process in validating cached resources for previously unvisited pages.

of Caddy that has been adapted according to the CACHECATALYST method. On the client side, we used Selenium [45] in conjunction with ChromeDriver 105 and Lighthouse to load the pages and measure performance metrics.

Reduction in the number of RTTs. We first examined the efficacy of CACHECATALYST in reducing the number of RTTs during page revisits. We initially loaded each page once to warm up the browser's cache, and then reloaded it at intervals of one minute, one hour, six hours, one day, and one week[5]. For each reload, we measured the number of RTTs in the critical path. This procedure was conducted for each page twice: once with CACHECATALYST disabled and once with CACHECATALYST enabled. Figure 6 illustrates the reduction in the number of RTTs after enabling CACHECATALYST. The results are presented both with and without the cache-aware push feature enabled. Given that the number of RTTs varies across pages, we calculated the RTT reduction for each page as a percentage and present the distribution of these values. The least median point is at the value 53%, indicating that for half of the pages, the number of RTTs during revisits decreased by more than 53%. The mean absolute reduction in the number of RTTs across all pages was 4.3.

We then examined the reduction in RTTs when navigating to new pages of a previously visited website. To this end, for each website, we first cleared the browser's cache, then loaded one of its pages, and subsequently loaded its 9 other pages consecutively at one-minute intervals. As shown in Figure 6, the number of RTTs is significantly reduced in this scenario as well. Nonetheless, the reduction in RTTs is lower in this

[5]To simulate the time delays, we advance the system clock.

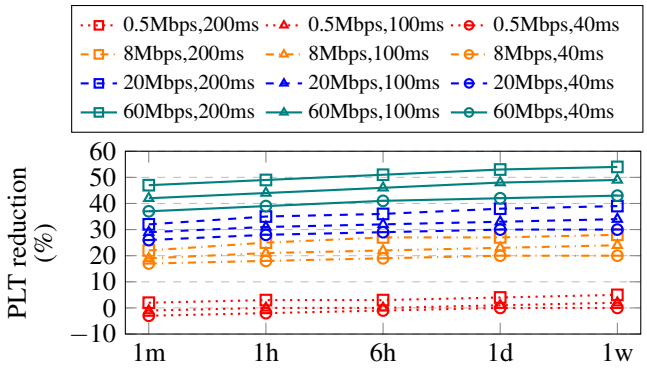
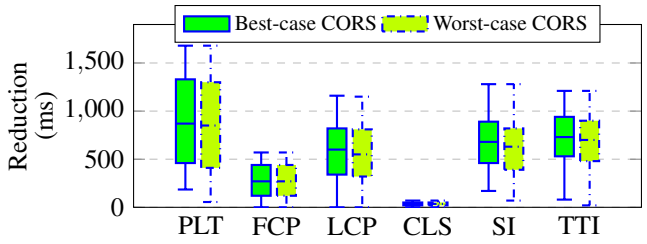


Figure 8: PLT reduction achieved by CACHECATALYST under different network conditions (the legend denotes **end-to-end** throughput and latency).

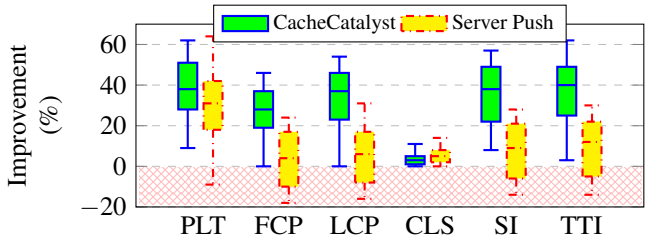
scenario because a smaller proportion of cached resources for those pages have been validated during the early cache validation phase. Figure 7 illustrates the coverage of the early cache validation process for previously unvisited pages, highlighting that a substantial portion of resources are shared across pages of the same website and can thus be efficiently reused from the browser cache through the early cache validation of CACHECATALYST. It is worth noting that resources missed at this stage can still be served from the browser cache using the traditional conditional `GET` request, but with one extra RTT.

Evaluation under various network conditions. Having established the reduction in the number of RTTs, we proceeded to examine how this translates to decreased PLT under various network conditions. To this end, we repeated the previous experiment under different network scenarios, varying both link throughput and latency. For each scenario, we measured PLT[6] with and without CACHECATALYST enabled. Figure 8 illustrates the median reduction in PLT achieved through CACHECATALYST without the cache-aware push feature enabled. While the reduction in PLT at low throughput is modest, a substantial decrease is observed at high throughput. Under high-throughput conditions, the primary bottleneck in web page loading shifts from bandwidth limitations to latency. Consequently, by eliminating unnecessary RTTs, the PLT is significantly reduced. Furthermore, at a constant throughput, the improvement in PLT becomes more pronounced as latency increases. It suggests that the proposed method offers greater benefits to users in regions geographically distant from web servers. As also evident from the figure, at a throughput of 0.5 Mbps, the additional metadata overhead introduced by CACHECATALYST leads to a degradation in performance. It is noteworthy that a throughput of 60 Mbps and a latency of 40 ms represent the median condition for global 5G Internet access [49], under which the PLT of different pages was reduced by 200 to 1700 ms. In the case where the cache-aware push was enabled, we observed results that were, on average,

[6]PLT measurement was performed using `OnLoad` event of the browser.



(a) Distribution of absolute performance improvements achieved by CacheCatalyst across pages (the CORS scenario is discussed in Section 7)



(b) Distribution of relative improvements across pages

Figure 9: Evaluation results for various performance metrics. The cache-aware push feature of CacheCatalyst was disabled.

12% better than the results shown in Figure 8.

User-centric performance metrics. To further assess the impact on user experience, we extended our analysis to five other performance metrics: First Contentful Paint (FCP), Largest Contentful Paint (LCP), Cumulative Layout Shift (CLS), Speed Index (SI), and Time to Interactive (TTI). These metrics, which can be easily measured using Lighthouse, are widely regarded as more representative of perceived user experience than PLT [54]. Numerous reports indicate that even a 100-ms improvement in these metrics can lead to a substantial increase in revenue [9, 50]. While these metrics are not solely determined by the speed of fetching resources—page design also plays an important role—our experiments show that more effective cache utilization can have a substantial positive impact on their values. We repeated the previous experiment under the following conditions: bandwidth and latency were set to 60 Mbps and 40 ms (the median condition for global 5G Internet access [49]), respectively. Page reloads were executed at 5-minute intervals, and the cache-aware push feature was not enabled. Figure 9 illustrates both the absolute and relative improvements achieved by CACHECATALYST for each of these performance metrics across pages.

FCP marks the time when the first content is displayed, and LCP indicates when the largest (main) visible element is rendered. Our results show substantial improvements in both, especially in LCP. The median improvement is 28% for FCP and 37% for LCP. The smaller gain for FCP is expected, as it depends on fewer resources. CLS, which measures unexpected layout shifts during loading, shows only

minor improvement, reflecting its stronger dependence on page design rather than network performance. In contrast, SI, which reflects how quickly visible content is populated, and TTI, which measures when the page becomes interactive, both exhibit notable improvements. These findings confirm that more effective cache utilization through the implementation of CACHECATALYST significantly enhances the web user experience, particularly by improving perceived visual completeness, smoothness, and interactivity.

Comparison with Server Push. Caddy supports the standard Server Push mechanism. To compare Server Push with CACHECATALYST, we enabled this feature in Caddy and provided the web server with the resource list of each page as resources to be pushed. We repeated the previous experiment for Server Push. The performance improvements achieved through Server Push are represented in Figure 9b. While Server Push achieves improvements in PLT comparable to CACHECATALYST, it performs significantly worse in the user-centric performance metrics. More importantly, in many cases, Server Push **degrades** these metrics (i.e., results in negative improvement values). These observations can be attributed primarily to the processing overhead incurred by the client browser when receiving all resources via Server Push. The reception and processing of these resources consume significant computational time. In contrast, when utilizing cached resources, the browser not only avoids the processing costs associated with receiving these assets but can also leverage the results of previous parsing and JavaScript executions. Previous research has also shown that Server Push can lead to performance degradation [61, 62]. Another negative aspect of Server Push is its traffic overhead. Figure 10 compares the traffic overhead of Server Push and CACHECATALYST, with CACHECATALYST's cache-aware push feature enabled. As shown, Server Push imposes substantially higher traffic overhead, which not only increases users' data costs but also introduces server-side bottlenecks. At the median point, its overhead is three orders of magnitude greater than that of CACHECATALYST. Consequently, a web server using Server Push can serve far fewer clients simultaneously compared to one employing CACHECATALYST. These are the very reasons that led to the deprecation of Server Push [12, 30]. However, CACHECATALYST avoids these drawbacks. As a final point in our evaluation, we examined the CPU and memory utilization of CACHECATALYST on client and server processes under varying request rates. Compared to execution without CACHECATALYST, the difference was negligible, with average CPU and RAM variation below 1%.

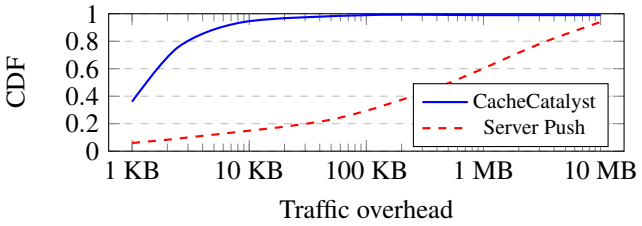


Figure 10: Traffic overhead of CACHECATALYST and Server Push. CACHECATALYST's overhead results from cache validation information for cached resources, while Server Push overhead comes from transmitting resources already present and usable in the browser's cache. The median overhead of CACHECATALYST and Server Push is about 1.5 KB and 1 MB, respectively, highlighting the substantially higher overhead of Server Push. (Server Push truncated at P94)

7 Discussion

In this section, we discuss the limitations of CACHECATALYST and their implications. The most significant limitation is that when navigating to a new page on a previously visited website (not revisiting a page), some cached resources that could be used might not be validated during the early cache validation. However, as Figure 7 shows, in most cases, the majority of these resources are included in the early cache validation process. Nevertheless, the resource coverage of the early cache validation for loading new pages should be improved, and this could be a potential research topic for future work. It is worth noting that the resources missed in early cache validation can still be retrieved from the cache using the traditional conditional `GET`.

Another important limitation arises from constraints imposed by CORS. For some cross-origin resources that are fetched by scripts, cache validation responses may not be returned due to CORS restrictions, depending on the CORS configuration of the corresponding websites. Accurately modeling this behavior would require testing, on a per-resource basis, whether each origin server permits responding to cross-origin cache validation requests from an unknown origin. Performing such fine-grained testing is impractical. To account for this issue while maintaining realistic experimental conditions, we conducted an additional experiment that assumes a worst-case scenario. In this experiment, for all cross-origin resources fetched by scripts, we conservatively disable cache state updates, effectively treating these resources as if their cache validation requests were rejected by the origin servers. The resulting improvements in performance metrics under this pessimistic assumption can be seen in Figure 9a. Although the performance improvements are somewhat smaller compared to our earlier experiments, CACHECATALYST still achieves substantial gains. It is important to note that CORS restrictions apply only to cross-origin resources fetched via scripts, not to cross-origin resources embedded directly in HTML. Moreover, our evaluation assumes a conservative worst-case scenario: in practice, many cross-origin resources are not restricted by CORS and can be fetched from arbitrary origins. Typically, resources that contain sensitive information enforce strict CORS policies and are excluded from caching by design, making CACHECATALYST's applicability to them

irrelevant.

Other limitations stem from our Service Worker–based implementation. While some browsers may impose restrictions on Service Workers, all major browsers support them today. Websites that already use Service Workers for other purposes must ensure compatibility with our caching logic, which is manageable with careful design. Another limitation is that Service Workers cannot share caches across domains; however, identical cross-site resources are uncommon and rarely part of a site's core content. This limitation does not apply to subdomains of the same website, which can share a single Service Worker. We note that if this new caching approach is adopted as part of the HTTP standard and receives native support from browsers, these minor issues will no longer exist.

Finally, we consider the case of websites served through a CDN. This setup requires no special modifications beyond deploying CACHECATALYST on CDN servers in addition to the origin. When a browser request reaches the CDN, the CDN validates resources it can account for locally. For resources requiring verification with the origin, the CDN acts as a client to the origin server, carrying out validation in the same CACHECATALYST manner.

8 Related Work

Cache-related methods. Since many web publishers do not configure the cache-control headers correctly, Raza *et al.* [42] have proposed Extreme Cache, a cache proxy between clients and servers that sets the cache headers by estimating the change rate of objects. However, as mentioned, estimating the change time of a resource is not straightforward, and this paper does not provide any report on the estimation accuracy. Moreover, the change time of many resources is not predictable at all, in which case the `no-cache` directive is used. Wang *et al.* [56] demonstrated that changes in the content of resources are often limited, with large portions remaining constant. Based on this, they proposed Micro-caching method, which allows for caching only the parts of a resource (such as JavaScript or CSS statements) that do not change, instead of the entire resource. This way, only the changed parts of each resource are sent to the client. This method reduces the download time of these resources. Fawkes [28], a server-side module, captures all content that remains unchanged across versions of a page's top-level HTML and uses it to create a static template. This template is cached on the client side. In subsequent client visits to the web page, the server generates dynamic patches that express the updates (i.e., DOM transformations) required to convert the template page state into the latest version of the page. Rewap [25] acts in a similar manner. This method does not eliminate the RTT of unpredictable dynamic resources. Geol *et al.* [11] proposed a browser-based solution that extends the cache of browsers to reuse identical computations (e.g., JavaScript execution) from prior page loads. This method is orthogonal to our work, and an optimal caching approach can further augment the benefits of using an execution cache.

Remote dependency resolution. An effective approach to resolving the problem of last-mile latency is to deploy a remote dependency resolution (RDR) proxy [35, 47, 48]. An RDR proxy is implemented using a headless browser on a cloud server with low-latency network paths to origin web servers. It moves the resolution of resource dependencies closer to web servers and performs resource fetches on behalf of the client. After fetching all resources of the requested web page, it sends the entire web page in bulk to the client. However, this method poses security and privacy issues since it acts as a man-in-the-middle for TLS connections. This not only violates end-to-end TLS security but also requires clients to provide proxies with the clear form of their private HTTP cookies. WatchTower [36] is a workaround for this problem that requires each HTTPS origin to run its own RDR proxy, which is expensive. It is also noteworthy that some proxies perform additional optimizations, such as minification and compression, to accelerate resource downloading and reduce mobile data usage [1, 46].

Server push. Numerous studies have shown that HTTP/2 Server Push [4] can result in sending resources that are either not needed or already cached by the client [55, 60]. Additionally, it can increase the page load time due to the increased computational load on the client side [61, 62]. It has also been shown that network performance characteristics play a major role in the effectiveness of Server Push [43]. Klotski [6], Shandian [57], and WebGaze [23] identify high-priority resources of pages in terms of user utility and prioritize pushing them. Another problem with Server Push is that the content of many web pages is often served by multiple domains, and the main web server cannot securely push resources from other domains. Hence, some authors have proposed providing the client with hints in the form of URLs for resources that the client should fetch before its own dependency resolution [15, 33, 44].

9 Conclusion

We showed that the current web caching design is ill-suited to today's Internet, where bottlenecks stem from latency rather than bandwidth. In particular, the cache re-validation mechanism of HTTP imposes unnecessary RTTs that degrade web performance. To address this, we proposed CACHECATALYST, which removes these RTTs, eliminates the need to configure resource TTLs, avoids conservative cache usage, and provides optimized server push. Our Service Worker–based implementation requires no browser modifications, making deployment practical. Evaluation results show substantial improvements across key performance metrics (PLT, FCP, LCP, SI, and TTI), confirming the effectiveness of this new web caching design.

References

[1] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 367–380, 2015.

[2] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M Maggs. On landing and internal web pages: The strange case of jekyll and hyde in web performance measurement. In *Proceedings of the ACM Internet Measurement Conference (IMC 20)*, pages 680–695, 2020.

[3] Alemnew Sheferaw Asrese, Steffie Jacob Eravuchira, Vaibhav Bajpai, Pasi Sarolahti, and Jörg Ott. Measuring web latency and rendering performance: Method, tools, and longitudinal dataset. *IEEE Transactions on Network and Service Management*, 16(2):535–549, 2019.

[4] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext transfer protocol version 2 (http/2). Technical report, IETF RFC 7540, 2015.

[5] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC 11)*, pages 313–328, 2011.

[6] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V Madhyastha, and Vyas Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 439–453, 2015.

[7] Caddy. Caddy source code, 2024. https://github.com/caddyserver/caddy.

[8] Caddy. Caddy web server, 2024. https://caddyserver.com.

[9] Chrome DevRel. Milliseconds make millions, 2020. https://web.dev/case-studies/milliseconds-make-millions.

[10] Fastcompany. How one second could cost amazon $1.6 billion in sales, 2012. https://tinyurl.com/4rck9z3r.

[11] Ayush Goel, Vaspol Ruamviboonsuk, Ravi Netravali, and Harsha V Madhyastha. Rethinking client-side caching for the mobile web. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 112–118, 2021.

[12] Google. Remove http/2 server push from chrome, 2022. https://developer.chrome.com/blog/removing-push/.

[13] Google. Lighthouse, 2024. https://github.com/GoogleChrome/lighthouse.

[14] Jake Brutlag (Google). Speed matters for google web search, 2009. https://services.google.com/fh/files/blogs/google_delayexp.pdf.

[15] Bo Han, Shuai Hao, and Feng Qian. Metapush: Cellular-friendly server push for http/2. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 57–62, 2015.

[16] A Hesam Mohseni, Amir Hossein Jahangir, and Seyed Mohammad Hosseini. Toward a comprehensive subjective evaluation of voip users' quality of experience (qoe): a case study on persian language. *Multimedia Tools and Applications*, 80(21):31783–31802, 2021.

[17] Mohammad Hosseini, Sina Darabi, Patrick Eugster, Mahmood Choopani, and Amir Hossein Jahangir. Rethinking web caching: An optimization for the latency-constrained internet. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pages 326–334, 2024.

[18] Mohammad Hosseini, Sina Darabi, Amir Hossein Jahangir, and Ali Movaghar. Yuz: Improving performance of cluster-based services by near-l4 session-persistent load balancing. *IEEE Transactions on Network and Service Management*, 21(2):1929–1942, 2023.

[19] Mohammad Hosseini, Sina Darabi, Hannaneh B Pasandi, Mohammad Nakhjiri, and Patrick Eugster. Application-driven reexamination of datacenter microbursts. In *Abstracts of the 2025 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 28–30, 2025.

[20] Mohammad Hosseini, Sina Darabi, Hannaneh B Pasandi, Mohammad Nakhjiri, and Patrick Eugster. Poison comes in small packages: Application-driven reexamination of datacenter microbursts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(2):1–23, 2025.

[21] httparchive. State of the web, 2025. https://httparchive.org/reports/page-weight?start=2025_01_01&end=2025_07_01&view=list.

[22] Sunghwan Ihm and Vivek S Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC 11)*, pages 295–312, 2011.

[23] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R Das. Improving user perceived page load times using gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 545–559, 2017.

[24] Rashna Kumar, Sana Asif, Elise Lee, and Fabián E Bustamante. Each at its own pace: Third-party dependency and centralization around the world. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS 23)*, 7(1):1–29, 2023.

[25] Xuanzhe Liu, Yun Ma, Shuailiang Dong, Yunxin Liu, Tao Xie, and Gang Huang. Rewap: Reducing redundant transfers for mobile web browsing via app-specific resource packaging. *IEEE Transactions on Mobile Computing*, 16(9):2625–2638, 2016.

[26] Xuanzhe Liu, Yun Ma, Yunxin Liu, Tao Xie, and Gang Huang. Demystifying the imperfect client-side cache performance of mobile web browsing. *IEEE Transactions on Mobile Computing*, 15(9):2206–2220, 2016.

[27] Yun Ma, Xuanzhe Liu, Shuhui Zhang, Ruirui Xiang, Yunxin Liu, and Tao Xie. Measurement and analysis of mobile web cache performance. In *Proceedings of the 24th International Conference on World Wide Web*, pages 691–701, 2015.

[28] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. Fawkes: Faster mobile page loads via app-inspired static templating. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 879–894, 2020.

[29] Gonzalo Martínez, José Alberto Hernández, Pedro Reviriego, and Paul Reinheimer. Round trip time (rtt) delay in the internet: Analysis and trends. *IEEE Network*, 38(2):280–285, 2023.

[30] Mozilla. Firefox 132.0 release notes, 2024. https://www.mozilla.org/en-US/firefox/132.0/releasenotes/.

[31] Mozilla. Cross-origin resource sharing (cors), 2025. https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS.

[32] Javad Nejati and Aruna Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*, pages 1305–1315, 2016.

[33] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[34] Ravi Netravali and James Mickens. Remote-control caching: Proxy-based url rewriting to decrease mobile browsing bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, pages 63–68, 2018.

[35] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate record-and-replay for http. In *2015 USENIX Annual Technical Conference (ATC 15)*, pages 417–429, 2015.

[36] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. Watchtower: Fast, secure mobile page loads using remote dependency resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 430–443, 2019.

[37] Ookla. Speedtest global index: Ranking mobile and fixed broadband speeds from around the world, 2025. https://www.speedtest.net/global-index.

[38] Opensignal. How at&t, sprint, t-mobile and verizon differ in their early 5g approach, 2020. https://tinyurl.com/5n73m8jk.

[39] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 19)*, pages 1–15, 2019.

[40] Feng Qian, Kee Shen Quah, Junxian Huang, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Web caching on smartphones: ideal vs. reality. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 127–140, 2012.

[41] Murali Ramanujam, Harsha V Madhyastha, and Ravi Netravali. Marauder: synergized caching and prefetching for low-risk mobile app acceleration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 350–362, 2021.

[42] Ali Raza, Yasir Zaki, Thomas Pötsch, Jay Chen, and Lakshmi Subramanian. Extreme web caching for faster web browsing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM 15)*, pages 111–112, 2015.

[43] Sanae Rosen, Bo Han, Shuai Hao, Z Morley Mao, and Feng Qian. Push or request: An investigation of http/2

server push for improving mobile performance. In *Proceedings of the 26th International Conference on World Wide Web*, pages 459–468, 2017.

[44] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V Madhyastha. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 17)*, pages 390–403, 2017.

[45] Selenium. A browser automation framework, 2024. https://www.selenium.dev.

[46] Shailendra Singh, Harsha V Madhyastha, Srikanth V Krishnamurthy, and Ramesh Govindan. Flexiweb: Network-aware compaction for accelerating mobile web transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 604–616, 2015.

[47] Ashiwan Sivakumar, Chuan Jiang, Yun Seong Nam, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Sanjay G Rao, Subhabrata Sen, Mithuna Thottethodi, and TN Vijaykumar. Nutshell: Scalable whittled proxy execution for low-latency web over cellular networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 448–461, 2017.

[48] Ashiwan Sivakumar, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, and Subhabrata Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*, pages 325–336, 2014.

[49] speedcheck. Global 5g speed index, 2024. https://www.speedcheck.org/5g-index/.

[50] WPO stats. The impact of web performance optimization (wpo) on user experience and business metrics, 2025. https://wpostats.com/.

[51] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, and Nazanin Magharei. Measuring and mitigating web performance bottlenecks in broadband access networks. In *Proceedings of the 2013 conference on Internet measurement conference (IMC 13)*, pages 213–226, 2013.

[52] tranco list. A research-oriented top sites ranking, 2025. https://tranco-list.eu.

[53] World Wide Web Consortium (W3C). Service workers, 2022. https://www.w3.org/TR/service-workers/.

[54] Philip Walton. User-centric performance metrics, 2023. https://web.dev/articles/user-centric-performance-metrics.

[55] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is spdy? In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 387–399, 2014.

[56] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. How much can we micro-cache web pages? In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC 14)*, pages 249–256, 2014.

[57] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up web page loads with shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 109–122, 2016.

[58] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. Why are web browsers slow on smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 91–96, 2011.

[59] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. How far can client-only solutions go for mobile browser speed? In *Proceedings of the 21st international conference on World Wide Web*, pages 31–40, 2012.

[60] Torsten Zimmermann, Jan Rüth, Benedikt Wolters, and Oliver Hohlfeld. How http/2 pushes the web: An empirical study of http/2 server push. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9. IEEE, 2017.

[61] Torsten Zimmermann, Benedikt Wolters, and Oliver Hohlfeld. A qoe perspective on http/2 server push. In *Proceedings of the Workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 1–6, 2017.

[62] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, and Klaus Wehrle. Is the web ready for http/2 server push? In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 13–19, 2018.