# Speed Kit: A Polyglot & GDPR-Compliant Approach For Caching Personalized Content

Wolfram Wingerath*, Felix Gessert*, Erik Witt*, Hannes Kuhlmann*,
Florian Bücklers*, Benjamin Wollmer†, and Norbert Ritter†
*Baqend GmbH, Stresemannstraße 23, 22769 Hamburg, Germany
{ww, fg, ew, hk, fb}@baqend.com
†University of Hamburg, Databases and Information Systems, Vogt-Kölln-Straße 30, 22527 Hamburg, Germany
{wollmer, ritter}@informatik.uni-hamburg.de

***Abstract*—Users leave when page loads take too long. This simple fact has complex implications for virtually all modern businesses, because accelerating content delivery through caching is not as simple as it used to be. As a fundamental technical challenge, the high degree of personalization in today's Web has seemingly outgrown the capabilities of traditional content delivery networks (CDNs) which have been designed for distributing static assets under fixed caching times. As an additional legal challenge for services with personalized content, an increasing number of regional data protection laws constrain the ways in which CDNs can be used in the first place. In this paper, we present Speed Kit as a radically different approach for content distribution that combines (1) a polyglot architecture for efficiently caching personalized content with (2) a natively GDPR-compliant client proxy that handles all sensitive information within the user device. We describe the system design and implementation, explain the custom cache coherence protocol to avoid data staleness and achieve $\Delta$-atomicity, and we share field experiences from over a year of productive use in the e-commerce industry.**

***Index Terms*—Web Caching, Cache Coherence, CDNs, Dynamic Data, Personalized Content, Data Privacy, Polyglot Storage**

## I. Motivation: Challenges in Modern Web Caching

HTTP caching plays a critical role in achieving high scalability and low latency for online presences across all industries. However, there are two critical challenges that remain unaddressed even by state-of-the-art content delivery networks (CDNs). The first major challenge is the **latency-staleness trade-off** that arises from the purely expiration-based HTTP caching model. While CDNs often support custom mechanisms for cache invalidation (e.g. Fastly [36], Akamai [39], Cloudflare [24]), there is no standardized procedure for retracting cached copies of a data item: Once an item is stored in the browser cache, for instance, the content provider has no means of updating it before the initially provided time-to-live (TTL) runs out. This is in stark contrast to virtually all modern Web applications that tailor content to the individual users in the form of product recommendations based on recently bought items or updates on friends' activities. To avoid staleness for users, personalized content is therefore often excluded from caching in practice, giving rise to latency stragglers and performance bottlenecks. Immutable and generic resources like images or stylesheets are thus often accelerated with a CDN, whereas the performance-critical website itself (i.e. the HTML) is typically considered uncacheable due to personalization and therefore delivered by the origin server. The second critical open challenge is related to the **legal ramifications** of using a CDN, since routing all incoming user traffic through it is mandatory for deployment. Since this effectively grants the CDN provider full access to information that is protected by regulations such as the General Data Protection Regulation (GDPR) [15] or the California Consumer Privacy Act of 2018 (CCPA) [9], employing a CDN requires careful consideration to avoid hefty fines [27] in case of non-compliance or data breaches.

To address the above issues, we propose Speed Kit as an approach for website acceleration that (1) caches personalized content without increasing staleness and (2) comes with a GDPR-compliant default configuration that makes sure protected data never leaves the user device. In Section II, we explain how Speed Kit works in principle and where the smart client proxy interacts with the polyglot backend, before we cover what optimizations are applied by default in Section III. In the following Section IV, we present the concept of Dynamic Blocks that enables Speed Kit to cache even personalized websites without delivering stale content. We subsequently focus on cache coherence in Section V, shedding light on the polyglot backend that combines CDN and database features behind a unified API, detects and invalidates stale caches, and thus enables $\Delta$-atomicity freshness guarantees. We then illustrate Speed Kit's mode of operation with an example of a typical page load in Section VI, compare our system to other approaches for accelerating Web content in Section VII, and finally conclude with a brief summary and outline of future work in Section VIII.

## II. Speed Kit: System Overview

Speed Kit is designed as a code plugin that essentially turns an existing website into a sophisticated Progressive Web App (PWA) [26]. It is engineered on top of the **Service Worker** specification [29] and is included as a single-line JavaScript code snippet. Speed Kit offers typical PWA features such as push notifications and an offline mode, but more importantly implements a unique caching strategy for combining low access latency with guaranteed data freshness.

**The Smart Client Proxy.** Figure 1 illustrates the principle

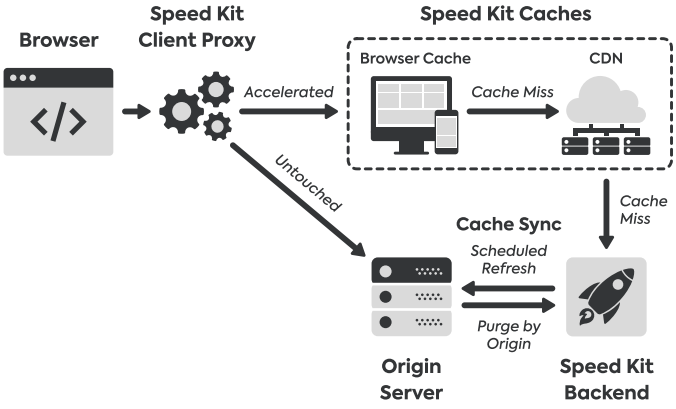


Fig. 1: To accelerate content delivery, the Speed Kit client proxy intercepts requests made by the browser and reroutes them: Instead of loading content from the original source, the browser fetches data from Speed Kit's fast caching infrastructure which is kept in-sync with the original data source in the background.

by which Speed Kit accelerates websites. Whenever a user visits the website, the Speed Kit client proxy is launched as a Service Worker process that runs concurrently to the main thread in the browser. As soon as the Service Worker is installed[1] and active, the client proxy starts intercepting all HTTP requests made by the browser to do one of two things: The proxy either (1) executes the request exactly as intended by the browser or it (2) rewrites the request to load the content from Speed Kit's own **caching infrastructure** instead of the original domain. In case of a cache hit, the rewritten request returns instantly (browser cache), within few milliseconds (CDN or other intermediate cache), or a few dozen to a few hundred milliseconds (Speed Kit backend), depending on the location and network connection of the client. On a cache miss, in contrast, the request is served as fast as the origin allows as the Speed Kit backend fetches the original response from the origin server and streams it to the requesting client through the caching hierarchy, thus making sure the response will be cached from this point on to accelerate subsequent requests. As the backend assigns TTLs independent of the ones provided by the origin, caching efficiency improves over time as Speed Kit learns from the observed workload. Our current approach is backward-oriented [20, Sec. 4.2], but we intend to explore predictive approaches in the future (cf. Section VIII).

**Staleness-Free Browser Caching.** Without special precautions, the danger of accessing stale data is inherent to using purely expiration-based caches like the browser cache, forward proxies, or ISP caches. To provide staleness guarantees independently of caching headers, Speed Kit implements an out-of-band cache invalidation mechanism. Since a comprehensive discussion of this topic would be out of scope, we only cover the basic procedure here and refer to the PhD thesis on Speed Kit's cache coherence protocol [17] for details. In essence, Speed Kit's backend keeps all cached data items in-sync with the original website as long as their TTLs are valid. To this end, we introduce a mechanism called **refresh** (cf. Section V) that crawls the origin, compares the current with the cached version of every asset, and purges stale cache entries either on a scheduled basis (refresh cron jobs) or in realtime (refresh API hooks). Whenever an inconsistency between cache and origin is detected, all invalidation-based caches are purged (e.g. the CDN); since purely expiration-based caches cannot be purged, however, the backend maintains all stale cache entries for their remaining TTL in a space-efficient data structure called the **Cache Sketch** [19] which the client retrieves in fixed intervals (default: 30 sec.). Since the Cache Sketch is based on Bloom filters [6] [8] and inherits the property to never produce false negatives for containment checks, the client proxy can safely leverage expiration-based caches by only using them for items not present in the Cache Sketch. We discuss the relationship between the Cache Sketch refresh interval and worst-case staleness in Section V.

**GDPR Compliance & Security.** Since personally identifiable information is neither processed nor stored by its backend, Speed Kit complies with data protection laws such as the GDPR unless explicitly configured otherwise. By design, Speed Kit only accelerates GET requests by caching their responses, i.e. POST, PUT, and DELETE requests are always processed by the original site alone. Since third-party cookies and credentials are further not accessible for Service Workers by specification [40], private data is hidden from Speed Kit unless exposed through GET requests (e.g. username and password visible in the URL). To avoid caching sensitive information by accident, Speed Kit only handles website requests that are explicitly whitelisted, excluding requests that are blacklisted; the client proxy's behavior is determined by a website-specific configuration that is transmitted to the browser when installing the Service Worker. To prevent man-in-the-middle or other attacks, Speed Kit further rejects websites that are not protected by **TLS encryption**. This is in line with Service Workers (and certain other browser features [30]) which are also exclusively available on TLS-secured websites.

## III. Default Optimizations

Apart from the features mentioned in Section II, Speed Kit provides a number of significant default optimizations which will be covered in the following.

**Implicit Network Tuning.** Each of Speed Kit's CDN nodes maintains a pool of persistent TLS connections to its backend. By establishing a TLS connection to the nearest CDN edge node, a client thus also immediately leverages the warm connection to Speed Kit's backing store. Since the initial TLS handshake takes at least two roundtrips [25, Ch. 4], establishing a secure connection with a CDN node in close proximity can also provide a substantial latency benefit over going all the way to the original web server. By employing OCSP stapling, stateless session resumption, dynamic record sizing, and several TCP tweaks (cf. [25, Ch. 4]), Speed Kit further ensures that the handshake never takes more than the minimum of two round trips to the nearest CDN node.

[1]To facilitate an early Service Worker activation, the code snippet should be included as early as possible, ideally in the website's header.

**Caching Third-Party Content.** In practice, page load time is often hampered by third-party dependencies such as social media integrations (e.g. Facebook), tracking providers (e.g. Google Analytics), JavaScript library CDNs (e.g. CDN.js), image hosters (e.g. eBay product images), or other service APIs (e.g. Google Maps). Since these resources are hosted on external domains which are out of the website owner's control, they are not eligible for CDN caching. In consequence, these additional third-party libraries introduce their own handshakes at TCP and TLS levels and are therefore loaded over cold low-bandwidth connections (cf. TCP slow start [41]). As Speed Kit is able to rewrite requests within the client, it fetches third-party resources through the established and fast HTTP/2 connection to Speed Kit's infrastructure rather than through slow connections to third-party domains. By keeping cached third-party resources up-to-date through periodic refresh jobs (cf. Section V), this optimization can be employed without additional staleness.

**Implicit Compression & Content Reencoding.** In order to alleviate the network footprint for loading complete scripts and HTML files, Speed Kit applies Gzip compression to all text-based resources that are delivered uncompressed by the original Web server. Oftentimes, however, the browser cache already contains a cached copy that is substantially similar to the up-to-date version of a resource at the server. In this case, transmitting the full resource means sending largely redundant information over the network irrespective of whether compression is enabled or not. If delta encoding [31] were supported for obtaining the up-to-date version, in contrast, the client could reuse portions of the (outdated) copy in the browser cache by only loading a diff and applying it locally. But in spite of standardization efforts and significant potential benefits [32], we are not aware of a single commercially relevant implementation of delta encoding in HTTP. We are currently evaluating an implementation of delta encoding within Speed Kit to increase efficiency of communication between the client proxy and the caching infrastructure.

**Dynamic Image Transcoding.** As another means to minimize page weight, Speed Kit transcodes images on-the-fly to the most efficient formats supported by the user's browser (e.g. WebP and Progressive JPEG) and rescales them to fit the requesting client's screen: A user with a high-resolution display will thus receive high-resolution images, while a user with an old mobile phone will receive a version that is natively scaled to the smaller screen dimensions. Speed Kit is further able to apply stronger compression to images when the network connection is poor in order to trade image quality against a faster paint. While mostly imperceptible for users, these optimizations lead to significant load time improvements, especially when images make up a large portion of the payload and bandwidth is limited (e.g. on mobile connections).

## IV. Dynamic Blocks: Accelerating Personalized Content Through Caching

Websites often contain customized elements, for example personal user greetings, product recommendations, or the number of items currently in the shopping cart. Since these are unique for every user and change frequently, caching them with traditional means is neither efficient (because of low cache hit rates) nor appropriate (because of potential staleness). Since Speed Kit's client proxy allows for custom processing within the user device, though, it is able to separate the generic website elements from the personalized ones during page load and merge them on arrival in the browser.

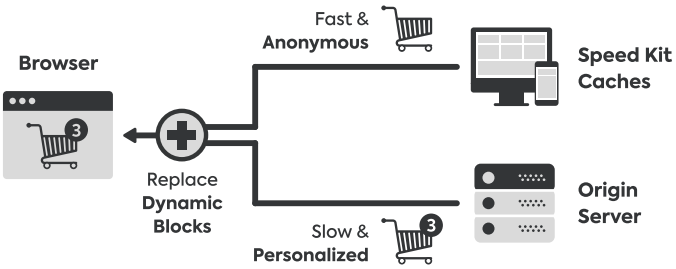


Fig. 2: With Dynamic Blocks, Speed Kit loads a cached anonymous website version first and fills in personalized info later.

Figure 2 illustrates the procedure for an e-commerce website with a customized shopping cart. On page load, the client proxy requests the personalized website from the origin server like the browser normally would, but in addition also requests the same page from Speed Kit's caches as a user that is *not logged in*. Since anonymous users are typically shown a generic version of the page, the *anonymous HTML* is easily[2] cacheable and can therefore be delivered faster (from Speed Kit's caches) than the personalized HTML (from the origin). As a result, the browser starts parsing the anonymous HTML, fetching linked assets, and rendering the page faster, too. When the personalized HTML from the origin server finally arrives, the client proxy injects the *personalized DOM elements* into the generic HTML. To this end, it first identifies[3] the relevant DOM elements – the **Dynamic Blocks** – in the anonymous and the personalized HTML files and then replaces them. A website with Dynamic Blocks thus behaves similar to a single-page app that loads a generic app shell from the cache and fetches personalized content asynchronously via Ajax/REST API requests.

## V. A Polyglot Backend For Flexible & Efficient Cache Maintenance

As described in Section II, the client proxy relies on the Speed Kit backend for discovering and invalidating stale caches through refreshing. The **refresh procedure** involves (1) selecting potentially stale cached assets from Speed Kit's backend storage, (2) downloading the corresponding assets from the origin server, (3) comparing the original and the cached versions, and (4) invalidating only those cached assets

[2]Speed Kit offers a customizable *HTML normalization* step to remove server-generated timestamps or other unique artifacts in the backend before putting the HTML into the caches.

[3]Since the client proxy currently does not recognize personalized sections on its own, Dynamic Blocks require specification of JavaScript query selectors in the Speed Kit config to pick the right DOM elements for content injection.

that have been updated. Compared to typical purge APIs known from CDNs, our approach has three significant benefits:

1) **No Unnecessary Invalidations**: By checking for actual content changes before clearing cache entries, even highly frequent refreshes do not affect cache hit rate when caches are still up-to-date.
2) **Query Expressiveness**: In addition to specification by URL or tag, the refresh API supports complex queries for selecting to-be-refreshed assets (e.g. a regex expression for URLs combined with a filter on the content type).
3) **Automatic Staleness Detection**: To facilitate easy cache synchronization with minimal integration effort, refreshes are scheduled for fully automatic content update checks in regular intervals via refresh cron jobs.

**Block Storage + Database = Scalability + Expressiveness.** To enable refreshes with query expressiveness on top of a CDN that only supports key-based purging by URL and tags, Speed Kit's backend stores a metadata summary for every cached asset in a database system (MongoDB) separately from the actual assets which are stored in a scalable block storage (S3). As the first step in every refresh job, the metadata store is queried to retrieve all summaries that match the provided refresh specification. Since every asset summary contains the asset URL and a content hash, the subsequent steps of the refresh job can be parallelized across multiple servers for the individual assets: downloading the current version from the origin, computing the content hash and comparing it against the one from the summary, and (only on actual change) updating the block and metadata storages, inserting the asset into the Cache Sketch, and purging the CDN. As an optimization, eTags [33, Sec. 14.19] are used for comparison when present to skip the expensive download and hash computation.

**Reliable & Fast Change Discovery.** The typical Speed Kit deployment is configured for a dual content discovery strategy: To make sure that Speed Kit's caches eventually reflect all and any content changes, all cached assets are crawled for changes on a regular basis through multiple **scheduled refresh** jobs (e.g. landing pages every 10 minutes, stylesheets once a day, and all cached assets every Thursday at 4 AM after deployment). For immediate synchronization after a content update, however, **real-time refresh** jobs can be triggered programmatically (e.g. through hooks in a content management system) or manually in the dashboard (e.g. by a developer after an unscheduled deployment). Since refreshes are executed in parallel on powerful machines with high-bandwidth network links, the duration of a refresh job is dominated by the time it takes the origin to serve the associated assets. The overhead for updating the caches is well below one second, as the time required for computing and comparing content hashes is negligible, the server-side Cache Sketch is updated immediately, and global propagation time for invalidations in Speed Kit's underlying CDN is only around 150ms [5].

**$\Delta$-Atomicity.** The actual staleness $\Delta_{max}$ exposed to the user is determined by three factors, namely (1) the time $t_{discovery}$ it takes Speed Kit's backend to detect content updates, (2) the change propagation time $t_{purge}$ for updating the CDN caches accordingly, and (3) the Cache Sketch refresh interval $t_{refresh}$ that specifies how often the client proxy checks for stale cache entries (cf. page 2):

$$\Delta_{max} \leq t_{discovery} + t_{purge} + t_{refresh}$$

While $t_{discovery}$ usually dominates staleness for content that is refreshed on a scheduled basis alone (e.g. once an hour or once a day), $t_{refresh}$ is the determining factor when real-time refreshes are used: We have observed $t_{purge} + t_{discovery} \ll 30sec$ to be a safe assumption in practice[4], so that $\Delta$-atomicity [21] with $\Delta = 60sec$ is easily satisfied with the default of $t_{refresh} = 30sec$.

## VI. Speed Kit in Action

Since mid-2018, Speed Kit has been used in e-commerce businesses ranging from small WordPress shops to large-scale international retailers such as the OTTO daughter Baur [34]. Through dynamic content optimization and its unique ability to accelerate personalized content, Speed Kit achieves a significant performance uplift even for website deployments that are already optimized. Especially for deployments with a slow time to first byte (TTFB), Speed Kit can bring down page loads from several seconds to below one second, thus often improving user engagement metrics such as the average session length or critical business KPIs like the conversion rate (see [18] for a concrete case study).

**A Page Load With Speed Kit.** The following end-to-end example describes a typical page load with Speed Kit to summarize and illustrate how its acceleration works:

1) **Compatibility check**: If Service Workers are supported[5], the browser runs the Speed Kit JavaScript snippet. Otherwise, the page loads without Speed Kit.
2) **Service Worker initialization**: Speed Kit is activated during the very first page load and starts serving requests from this point on. While the first requests of the very first page load are thus processed normally, every request is accelerated for returning visitors when the Service Worker is already installed.
3) **Request interception**: Any request issued by the browser is proxied through Speed Kit's Service Worker by default.
   - *Offline mode*: Speed Kit serves everything from cache when disconnected, but automatically synchronizes all content on reconnect.
   - *Normal operation*: Based on the config, Speed Kit decides whether to execute the unaltered request or to rewrite it to Speed Kit's caching infrastructure for acceleration and content optimization such as image transcoding and rescaling or implicit Gzip compression (cf. Section III).
4) **Staleness check**: To evaluate if expiration-based caches are safe to use, the client proxy looks up the requested

[4]As long as the origin serves content consistently fast, overall change discovery time with real-time refreshes is in the realm of single-digit seconds.

[5]Service Workers are supported by over 90% of all browsers [13].

resource in the local Cache Sketch which is retrieved once upfront and refreshed periodically (cf. Section II).

5) **Accelerated response**: If the Cache Sketch contains the resource, it might be stale and therefore must be requested from the nearest invalidation-based cache (i.e. CDN). Else, the response is served from the following caches:
   a) *Local cache*: If the requested response is present in one of the device-local caches (e.g. browser cache), Speed Kit returns it instantly.
   b) *CDN cache*: Upon a local cache miss, Speed Kit issues the request over the HTTP/2 connection to its CDN.
   c) *Speed Kit backend*: If there is a CDN cache miss, the response is fetched from one of Speed Kit's servers which (if it is not already available) fetches it from the original server. In the process, both asset and metadata (URL, content type, etc.) are stored in the polyglot backend (cf. Section V).
6) **Dynamic Blocks**: When requesting the HTML of a website configured for Dynamic Blocks (cf. Section IV), the client proxy concurrently retrieves an anonymous version from its fast caches and the fully personalized version from the website's origin. The anonymous version typically arrives faster and is therefore processed (and rendered) earlier. On receiving the personalized HTML, the client proxy extracts the personalized DOM tree elements and injects them into the rendered page, thus leaving it fully personalized.

## VII. Discussion & Competing Approaches

To expound where Speed Kit advances the current state of the art, we now contrast our own against other approaches for tackling Web performance.

**Speed Kit vs. CDNs.** Unlike traditional CDNs, Speed Kit does not only optimize content delivery between the website's origin and the CDN edge nodes, but goes all the way to the end user. Since its client proxy is located within the user device, Speed Kit is able to combine expiration-based browser caching with rigorous freshness guarantees while accelerating even personalized websites and third-party dependencies, all of which are out of reach for state-of-the-art CDN caching. Mechanisms like Fastly's Edge Side Includes [14], Amazon CloudFront's Lambda@Edge [3], and Cloudflare Workers [38] do allow custom processing on the edge, but they are complex to integrate into existing websites and do not address last-mile latency issues. Invalidating content that changes frequently is also often infeasible, since many CDNs provide purge latencies on the order of minutes (e.g. Google Cloud CDN [10], Amazon CloudFront [4], CDN77 [12]) or even hours (e.g. Cloudinary [7]).

Speed Kit's refresh jobs support complex queries for selecting *potentially stale* cache entries and only purge those that are actually stale. Modern CDNs, in contrast, are less expressive as they merely support purging individual assets by URL or all assets at once (e.g. CDN77 [12], StackPath [37], Yottaa [42]), or asset groups by tag, host, domain, or URL prefix (e.g. Cloudflare [11], Fastly [16], Akamai [1], Google Cloud CDN [22], Amazon CloudFront [2], KeyCDN [35]). We are not aware of a single purge API that provides query expressiveness comparable to that of Speed Kit's refresh API. Further, CDN purge APIs are less lenient than refreshes as they blindly invalidate all matching cache entries, irrespective of whether or not they are actually in-sync with the origin. By checking for actual changes before invalidation, refresh jobs are thus more convenient as they remove the need to identify the exact places where changes occur.

As another key distinction, Speed Kit does not process or store personally identifiable information like cookies or credentials, so that it can be deployed without violating GDPR compliance. Finally, Speed Kit is also easier to integrate into existing website than a typical CDN as it relies on including a script into the website's head rather than taking over the website's DNS.

**Speed Kit vs. Micro-Caching.** Many Web architectures employ micro-caching as a means to increase scalability: To fan out reads, they serve slightly stale resources through a tier of caching servers (e.g. Varnish [28]) in front of the actual web server. Micro-caching minimizes staleness through short TTLs by design, but thereby also causes frequent cache misses as cached responses expire quickly. This diminishes the efficiency of the caching tier and creates latency stragglers. Speed Kit does not have this problem, since its refresh and invalidation mechanisms decouple freshness guarantees from cache lifetimes. Even with large TTLs, staleness can be bounded by employing refreshes in periods (e.g. every 5 minutes) or in realtime (e.g. through CMS hooks): The browser cache will be used until it is invalidated by a refresh job, after which Speed Kit falls back to the CDN and other caches until expiration. Since all caches are backed by Speed Kit's storage backend, the origin is only hit on comparatively rare cache misses and during refreshes. It should also be noted that micro-caching is not applicable to personalized content, since short TTLs are ineffective (no cache hits), whereas large TTLs expose stale content to the user.

**Speed Kit vs. AMP & Instant Articles.** Google's Accelerated Mobile Pages (AMP) and Facebook's Instant Articles (IA) can also be used to create fast websites. Contrasting Speed Kit, however, AMP and IA do not work as plugin solutions for existing websites. Rather, they require rebuilding existing websites on Google's and Facebook's proprietary platforms, necessarily creating fast-loading pages as only features are allowed that do not hurt web performance.

AMP makes websites lean by enforcing limitations like restricted HTML (only a stripped-down version of HTML syntax), restricted JavaScript (no custom code allowed, except in iframes), restricted CSS (all stylings must be inlined and below 50 KB overall), no repaints (no resizing of DOM elements, i.e. only static sizes allowed), Google styling (mandatory Google bar at the top), and a stale-while-revalidate cache policy (i.e. users may see outdated content [23]). As another optimization, the browser prefetches the featured AMP pages already when displaying the Google search result. Since they

are thus loaded regardless of whether the user actually clicks the link, AMP pages can be rendered instantly. However, this only works for users visiting through the Google search and does not accelerate the page load for normal users. Instant Articles are technically similar to Google's AMP. One of the most significant distinctions, however, is that they are only accessible for Facebook users and therefore cannot be used to create public websites.

In contrast to AMP and IA, Speed Kit works for existing websites without restrictions on the target platform or the permitted HTML, CSS, or JavaScript constructs.

## VIII. Summary & Outlook

Speed Kit addresses both technical and legal challenges associated with content delivery in modern Web applications. First, it provides $\Delta$-atomicity freshness guarantees on top of traditional HTTP caching by combining refresh jobs for change discovery with a custom protocol for cache invalidation. Further, Speed Kit makes caching applicable to personalized websites that are typically considered uncacheable. To this end, it separates generic from user-tailored website elements through Dynamic Blocks and loads them individually. Finally, Speed Kit avoids data protection hazards by processing sensitive information exclusively on the user device, thus keeping it away from third parties such as CDN providers.

Since this paper describes our overall approach, several aspects are only covered briefly. In future publications, we therefore plan to present more details on semantics, trade-offs, and optimizations of the refresh procedure and the polyglot architecture supporting it. We further plan to evaluate advanced optimizations for content delivery (e.g. delta encoding) and explore different use cases for machine learning that we encountered in the context of our caching approach (e.g. predicting optimal TTLs on the basis of observed invalidation times).

## References


[1] Akamai Technologies. *Fast Purge API v3*, 2019. https://developer.akamai.com/api/core_features/fast_purge/v3.html (2019-06-06).

[2] Amazon Web Services, Inc. *Invalidation paths*, 2016. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Invalidation.html#invalidation-specifying-objects-paths (2019-06-06).

[3] Amazon Web Services, Inc. *Lambda@Edge*, 2019. https://aws.amazon.com/lambda/edge/ (2019-06-06).

[4] Amazon Web Services, Inc. *Why is CloudFront serving outdated content from Amazon S3?*, 2019. https://aws.amazon.com/premiumsupport/knowledge-center/cloudfront-serving-outdated-content-s3 (2019-06-06)

[5] H. Beheshti. Leveraging your CDN to cache "uncacheable" content. *Fastly Blog*, 2015. https://www.fastly.com/blog/leveraging-your-cdn-cache-uncacheable-content (2019-06-01).

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[7] O. Bogler. I've replaced an existing image with a new one, but my website still shows the old one. Why is that? *Cloudinary Support*, 2019. https://support.cloudinary.com/hc/en-us/articles/202520852 (2019-06-06).

[8] F. Bonomi, M. Mitzenmacher, R. Panigrahy, et al. An Improved Construction for Counting Bloom Filters. In *ESA*. Springer, 2006.

[9] California State Legislature. California Consumer Privacy Act of 2018. *California Civil Code*, 2018.

[10] Closte, LLC. *Google Cloud CDN*, 2018. https://closte.com/support/wordpress/google-cloud-cdn (2019-06-06).

[11] Cloudflare, Inc. *Purge Files by Cache-Tags or Host*, 2019. https://api.cloudflare.com/#zone-purge-files-by-cache-tags-or-host (2019-06-06).

[12] DataCamp, Ltd. *Data Management*, 2019. https://client.cdn77.com/support/api/version/2.0/data (2019-06-06).

[13] A. Deveria et al. Can I use Service Workers? *caniuse.com*, 2019. https://caniuse.com/#search=Service%20Workers (2019-07-13).

[14] C. Dixon. How do I enable basic ESI in my VCL? *Fastly Support*, 2014. https://support.fastly.com/hc/en-us/community/posts/360040447152 (2019-06-06).

[15] European Parliament and Council of the European Union. Regulation 2016/679 (General Data Protection Regulation). 2016.

[16] Fastly, Inc. *API Reference: Purging*, 2019. https://docs.fastly.com/api/purge.html (2019-06-06.

[17] F. Gessert. *Low Latency for Cloud Data Management*. PhD thesis, University of Hamburg, Von-Melle-Park 3, 20146 Hamburg, 2018.

[18] F. Gessert. AppelrathCüpper: How Speed Kit Helps in Fashion E-Commerce. *Baqend Tech Blog*, 2019. https://medium.com/p/c587f4c88ec (2019-05-25).

[19] F. Gessert, M. Schaarschmidt, W. Wingerath, et al. The Cache Sketch: Revisiting Expiration-Based Caching in the Age of Cloud Data Management. In *Proceedings of the BTW 2015*, 2015.

[20] F. Gessert, M. Schaarschmidt, W. Wingerath, et al. Quaestor: Query Web Caching for Database-as-a-Service Providers. *PVLDB*, 2017.

[21] W. Golab, X. Li, and M. A. Shah. Analyzing Consistency Properties for Fun and Profit. In *ACM PODC*, pages 197–206. ACM, 2011.

[22] Google. *Cache Invalidation Overview*, 2019. https://cloud.google.com/cdn/docs/cache-invalidation-overview (2019-06-06).

[23] Google. *Google AMP Cache Updates*, 2019. https://developers.google.com/amp/cache/overview#google-amp-cache-updates (2019-06-05).

[24] D. Grant. Introducing a Powerful Way to Purge Cache on CloudFlare: Purge by Cache-Tag. *Cloudflare Blog*, 2015. https://t.co/k6NHk3z0B6 (2019-05-25).

[25] I. Grigorik. *High Performance Browser Networking*. O'Reilly, 2013.

[26] D. A. Hume. *Progressive Web Apps*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.

[27] L. Kalman. New european data privacy and cyber security laws: One year later. *Commun. ACM*, 62(4):38–38, Mar. 2019.

[28] P.-H. Kamp. Varnish, 2019. https://varnish-cache.org/ (2019-05-27).

[29] M. Kruisselbrink, J. Song, A. Russell, and J. Archibald. Service workers. W3C working draft, W3C, 2017. https://www.w3.org/TR/2017/WD-service-workers-1-20171102/ (2019-05-26).

[30] V. Lynch. HTTPS-Only Features in Major Browsers. *The DigiCert Blog*, 2018. https://www.digicert.com/blog/https-only-features-in-browsers/ (2019-05-26).

[31] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta Encoding in HTTP. RFC 3229, RFC Editor, January 2002.

[32] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. *SIGCOMM Comput. Commun. Rev.*, 27(4):181–194, Oct. 1997.

[33] H. F. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, et al. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[34] Otto GmbH. Baur, 2019. https://www.baur.de/ (2019-07-12).

[35] Proinity LLC. *Purge Zone Tag*, 2019. https://www.keycdn.com/api#purge-zone-tag (2019-06-06).

[36] B. Spang. Building a Fast and Reliable Purging System. *Fastly Blog*, 2014. https://www.fastly.com/blog/building-fast-and-reliable-purging-system (2019-05-25).

[37] StackPath, LLC. *Purge cached content for all CDN sites on a stack*, 2019. https://developer.stackpath.com/en/api/cdn/#operation/PurgeContent (2019-06-06).

[38] K. Varda. Introducing Cloudflare Workers: Run JavaScript Service Workers at the Edge. *Cloudflare Blog*, 2017. https://blog.cloudflare.com/introducing-cloudflare-workers/ (2019-06-06).

[39] T. Vereecke. A Technical Deep Dive Into Purging by Cache Tag. *Akamai Developer Blog*, 2019. https://developer.akamai.com/blog/2019/03/28/technical-deep-dive-purging-cache-tag (2019-05-25).

[40] Web Hypertext Application Technology Working Group (WHATWG). *Fetch API Specification: Forbidden Header Names*. https://fetch.spec.whatwg.org/#forbidden-header-name (2019-06-04).

[41] J. Widmer, R. Denda, and M. Mauve. A Survey on TCP-friendly Congestion Control. *Netwrk. Mag. of Global Internetwkg.*, May 2001.

[42] Yottaa. *Flush Cache With Yottaa API*, 2019. https://support.yottaa.com/CustomerSupport/s/article/Flush-cache-with-Yottaa-API (2019-06-06).