Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly

Nikhil Kansal*, Murali Ramanujam*, Ravi Netravali
UCLA

Abstract

Despite their promise, HTTP/2's server push and preload features have seen minimal adoption. The reason is that the efficacy of a push/preload policy depends on subtle relationships between page content, browser state, device resources, and network conditions—static policies that generalize across environments remain elusive. We present Alohamora, a system that uses Reinforcement Learning to learn (and apply) the appropriate push/preload policy for a given page load based on inputs characterizing the page structure and execution environment. To ensure practical training despite the large number of pages served by a site and the massive space of potential policies to consider for a given page, Alohamora introduces several key innovations: a page clustering strategy that favorably balances push/preload insight extraction with the number of pages required for training, and a faithful page load simulator that can evaluate a policy in several milliseconds (compared to 10s of seconds with a real browser). Experiments across a wide range of pages and mobile environments (emulation and real-world) reveal that Alohamora accelerates page loads by 19-61%, provides 3.6-4× more benefits than recent push/preload systems, and properly adapts to *never* degrade performance.

1 Introduction

Mobile web browsing has rapidly risen in popularity [15, 17, 51]. Given the importance of mobile web speeds for both user satisfaction [6, 7, 19] and content provider revenue [18], a vast array of optimizations have been developed [8, 30, 39, 40, 43, 54, 57, 61]. Yet page loads remain too slow for users in practice, taking over 10 seconds to load even with state-of-the-art mobile devices and LTE cellular networks [4, 54].

Recent studies have identified that a key culprit to slow mobile page loads is the blocking network delays that arise from the dependencies between a page's objects [39, 54]. For example, a browser may learn that it needs an image only after fetching and executing a JavaScript file, which is discovered only after downloading and parsing the page's top-level HTML. Such dependency chains essentially serialize object fetches, leading to high load times, particularly in mobile settings where access link latencies tend to be high [23, 66].

The latest HTTP/2 standard [5] anticipated the negative impact of network delays on web performance, and in response, includes several relevant features. Most notable are HTTP/2 *push* and *preload*. With push, servers can proactively send objects to clients in anticipation of future requests; requests for already-pushed objects can be satisfied locally at the client, avoiding blocking network fetches. In contrast, with preload, servers can notify clients of objects that they will soon require (potentially from other domains) by listing those URLs in HTTP headers. Clients issue requests for those objects immediately after parsing HTTP headers, and without evaluating response bodies, thereby parallelizing network and computation tasks [54].

Unfortunately, despite their promise, developing performant push/preload policies has proven to be challenging, leading to low adoption rates. For example, we find that only 5% of the Alexa top 500 pages [3] include a domain that uses push or preload; this drops to 0.9% for the Alexa top 10,000 pages. A major reason is that the performance of a given push/preload policy depends on the subtle, low-level interactions between page content, browser (cache) state and execution dependencies, client device and network resources, and QoE goals [53, 60, 69, 70]. Consequently, even for a given page, we find that using a policy outside of the execution environment for which it was designed can either forego significant (18-31%) performance benefits or degrade performance by up to 20% compared to a default page load (§2).

These results preclude the use of the static policies and guidelines promoted by prior push/preload systems [53, 54, 70], and instead highlight the need for *dynamic, adaptive* policies that explicitly target the environments in which they are deployed. For example, the aggressive push/preload policies that effectively utilize resources in high-bandwidth settings must be shrunk or dispersed across a page load as link rates drop to avoid potential network contention that slows the downloads of blocking resources. Similarly, as device CPU speeds decrease, policies should grow in size to take advantage of the (increased) blocking compute delays that leave the network idle.

We present **Alohamora**, a system that learns and applies the appropriate push/preload policies for different pages and execution environments (Figure 1). Alohamora represents its policy generation logic as an expressive neural network that is trained offline using Reinforcement Learning (RL); we list the benefits of using RL in §3. During a client page load, Alohamora's model takes as input a set of features that summarize the client's execution environment (network, CPU, cache contents), and structural information about the page at hand, and outputs a push/preload policy intended to optimize QoE for the current load. Importantly, Alohamora *does not require new browser features*, and involves minimal server changes: servers provide structural information about their pages (which content management systems commonly track [14, 30, 67]), and Alohamora's policy generation runs transparently on a co-located frontend server.

* These authors contributed equally to this work.

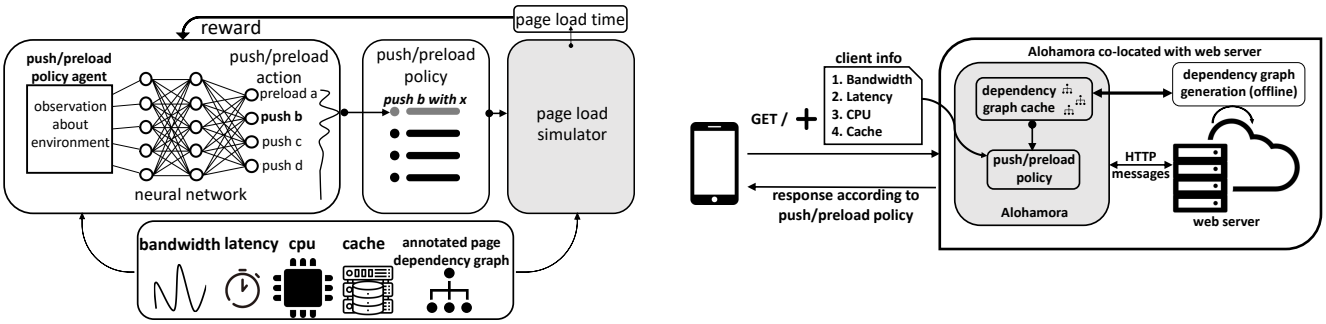


Figure 1: **Alohamora trains (left) its push/preload policy generation model using Reinforcement Learning, exploring a large search space of environments and policies, and learning from the resulting (simulated) performance. During client page loads (right), for each origin, Alohamora collects the required inference inputs from client browsers (using existing features) and servers which track changes to their page dependency graphs; the generated policies are applied transparently for the remainder of the page load.**

Realizing Alohamora's data-driven approach to HTTP/2 push/preload policy generation requires overcoming two key practical challenges with respect to training efficiency:

- **Generalizing across pages (§4)**: websites commonly serve thousands of pages, and it is impractical to incorporate each into the training process. However, failing to incorporate different pages during training may hide push/preload insights, and result in poorly generalizable models. To overcome this, Alohamora leverages our observation that even though sites serve thousands of URLs, their pages typically cover a far smaller number of page structures. The key idea is that these *shared structural properties typically dictate the efficacy of different push/preload strategies*. Thus, Alohamora needs not train on multiple pages with the same structural properties, as those would contribute similar push/preload insights. More specifically, push/preload benefits are dictated by resource utilization during the load process, which in turn can be characterized by 1) browser and inter-object dependencies, and 2) the overheads imposed by tasks involving the network and CPU. By extracting this structural information from a site's pages and clustering pages accordingly, we find that Alohamora is able to strike a desirable balance between the number of pages required for training and model generalizability.
- **Simulating page loads (§5)**: Alohamora's training involves testing different push/preload policies in diverse environments. However, the large number of potential environments and push/preload policies per page (exponential in terms of object count), coupled with the high mobile load times described above, make this approach far too slow. For example, even for a single environment, exploring the thousands of potential policies for `nytimes.com` would require 30 days on a powerful desktop machine. To handle this, Alohamora introduces a novel page load simulator which faithfully (errors of 0.4-2.2%) predicts the performance of a policy 3-4 orders of magnitude faster than running a real browser; for context, this cuts training time to 20 minutes for `nytimes.com`. To the best of our knowledge, Alohamora's simulator is the first to faithfully predict page load performance *across metrics and environmental conditions*, without requiring costly profiles [68] or emulation [60] for each environment. The key insight is in *judiciously extracting invariants about the page load process and superimposing variable resource constraints* by simulating browser-environment interactions; invariants (e.g., page/browser dependencies) are collected via a single profiling run with a real browser, while variable properties about the target environment and push/preload policy are taken as input. The simulator is general enough to support other optimizations that modulate network/compute delays [2, 40, 43, 57] or scheduling policies [8, 39].

We evaluated Alohamora using more than 500 web pages, and a wide range of mobile networks, client devices, and cache conditions. Our experiments, both emulation and real-world, reveal that Alohamora reduces page load time and Speed Index by 19-61% and 15-48%, respectively, compared to a default page load (i.e., no push/preload) and standard push/preload-all policy. In addition, Alohamora marginally (0.9-1.7×) outperforms WatchTower [43], a recent proxy-based accelerator, and delivers 3.6-4× more benefits than Vroom [54], a state-of-the-art push/preload system. Importantly, whereas Vroom slows down 24-34% of page loads, Alohamora properly adapts to *never degrade performance*. Source code and experimental data for Alohamora are available at https://github.com/nkansal96/alohamora.

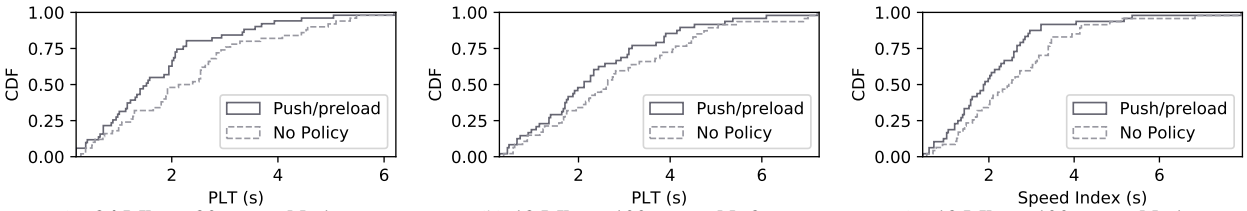
2 Background and Motivation

We begin with an overview of HTTP/2 (§2.1), and then present measurements that illustrate the potential benefits and challenges with HTTP/2 push and preload (§2.2).

2.1 HTTP/2 Overview

HTTP/2 [5] alters the traditional HTTP/1.1 page load process primarily by adding the following new features:

- **Request multiplexing:** With HTTP/1.1, browsers can open and reuse multiple concurrent TCP connections per origin. In contrast, HTTP/2 permits only a single con-



(a) **24 Mbps, 20 ms, cold, 1×** (b) **12 Mbps, 100 ms, cold, 2×** (c) **12 Mbps, 100 ms, cold, 1×**

Figure 2: **Push/preload benefits when policies are explicitly tuned to the deployment environment's available resources. Environments are listed as {bandwidth, latency, cache setting, CPU slowdown}. Results are shown for either Page Load Time (PLT) or Speed Index.**

nection per origin, and allows browsers to multiplex requests onto that connection as parallel *streams*. Unlike HTTP/1.1 pipelining, HTTP/2's multiplexing permits out-of-request-order delivery to avoid head of line blocking. HTTP/2 also mandates the use of TLS (and thus, HTTPS).

- **Server push:** Unlike HTTP/1.1 servers which only serve objects in response to explicit client requests, HTTP/2 servers can *push* objects that they own in anticipation of future requests. Servers have flexibility in defining a *push policy*, which specifies the mapping between objects that are explicitly requested and the set of files pushed along with them. Pushed objects are usable for the duration of the current page load, regardless of the associated HTTP caching headers. Note that pushed objects that are already in the browser's cache imply wasted network bandwidth.
- **Preload:** HTTP/2 also carried over HTTP/1.1's preload feature, which enables servers to list URLs to fetch directly in HTTP Link headers. Upon parsing such Link headers (i.e., before parsing the response body), browsers will immediately issue requests for the listed URLs; responses are not evaluated until the objects are referenced by the page. Thus, like push, preload enables servers to help browsers pre-warm their caches rather than relying on object execution to discover downstream objects. However, preload differs from push in that: 1) requests are client-driven and still involve network delays *to* origin servers, 2) the risk of re-downloading cached objects is eliminated since preload requests pass through the browser cache, and 3) servers can specify to preload third-party objects, not just objects that they own.
- **Stream prioritization:** HTTP/2 offers a mechanism with which both clients and servers can explicitly specify how parallel request streams on a single TCP connection share network and server-side processing resources. In particular, endpoints can annotate each request with a single integer that denotes its target share of the resources.

In this paper, we focus on the HTTP/2 push and preload features because they are configured by servers, i.e., Alohamora's target deployment location. In contrast, stream priorities are usually specified by browsers [11, 64], and yield limited benefits [31]. We note that push/preload policies exhibit a notion of prioritization that we do consider: "push A+B with C" and "push B+A with C" are different policies.

2.2 Limitations of Static Push/Preload Policies

Push/preload policies have been widely studied, yielding mixed performance results [16, 53, 54, 60, 70]. The key reason is that the performance of a policy depends on numerous page and environmental properties. To better understand the relationships between these properties and push/preload policies, we performed a study involving 50 random pages from the Alexa top 500 US sites [3]. Our results use the same methodology and environmental parameters (network, device CPU, cache, QoE metrics) described in §6.1.

For each environment and page combination, we selected the best policy using a brute force search. Since the number of potential policies for a page scales exponentially with the number of objects (which regularly exceeds 100), a complete brute force search across environments is impractical. Instead, to ensure practicality and sufficient coverage, we weighted object types based on their potential for blocking the client-side page load (i.e., JS = CSS > image > font) [39, 59]. To generate a policy, we randomly selected the number *N* of objects to push/preload, and then sampled the object types *N* times according to their relative priorities (picking randomly within each type). Finally, we randomly selected the fraction of objects to mark as push vs. preload, and for each object, we randomly selected an earlier object in the load to push/preload from. Using this approach, for each page, we generated 200 policies and picked the one that delivered the largest improvements.

Takeaway 1: Push/preload has potential. For each environment and page pair, we compared the best push/preload policy (selected explicitly for that pair) to a default page load (i.e., no push/preload). Figure 2 shows representative results for several settings. As shown, when selected explicitly based on the environment, push/preload policies are able to provide significant speedups. For instance, in the {24 Mbps, 20 ms RTT, cold cache, 1x CPU slowdown} setting, median (95th percentile) page load time benefits are 18% (44%).

Takeaway 2: Push/preload policies do not generalize well. Despite the potential benefits, our results also highlight that push/preload policies quickly degrade in performance when run outside of the precise environments for which they were tuned. To evaluate this, we performed multiple experiments in which we started with a fixed environment, and selectively modulated each environmental factor while keeping the oth-

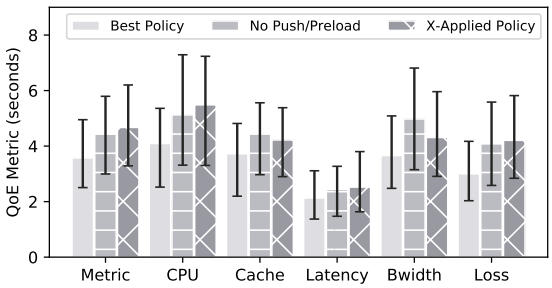


Figure 3: **Push/preload performance degrades as the environment changes. The base configuration was {12 Mbps, 100 ms, cold cache, 1x CPU, PLT}; each cluster modulates only one factor. "Best Policy" was tuned to each setting, and "X-Applied" applies the base configuration's best policy to each setting. Bars show medians, with error bars spanning 25-75th percentiles.**

ers fixed. In each resulting environment, we compared the performance of 1) the best policy for the fixed environment, 2) the best policy for the resulting environment, and 3) no push/preload. Figure 3 depicts our results for one fixed environment; we omit results for others due to space constraints, but note that the trends persist. These results illustrate two significant drawbacks to using push/preload policies across environments. First, they leave significant (18.4-30.7%) performance gains on the table compared to policies designed explicitly for the deployment setting. Second, and worse, they can degrade performance compared to a default page load. For instance, performance degrades by 6% (20%) at the median (95th percentile) when device CPU speeds change. These slowdowns are even more pronounced when multiple environmental factors are modified in parallel.

Summary: Collectively, our results suggest that, to realize the significant performance potential of push/preload, policies must be designed to explicitly consider page properties and characteristics of the target deployment environment.

3 Design Overview

Figure 1 shows the high-level design of Alohamora's offline training and online (i.e., during client page loads) inference phases. In this section, we will describe the workflow for each task in the context of a single web page. We present extensions to ensure practical training via cross-page generalization (§4) and page simulation (§5) in subsequent sections.

3.1 Offline Training

Why RL? Alohamora represents its push/preload policy generator as a neural network that is trained using Reinforcement Learning (RL) [32]. RL offers several advantages in this setting compared to more standard, supervised learning approaches. Most notably, the search space of push/preload policies is massive (exponential in terms of the number of objects in a page, which regularly exceeds 100), and it is impractical to generate a labeled training dataset that incorporates all of the fruitful push/preload policies for a page. RL overcomes this by using an efficient exploration strategy, whereby experience of prior tested policies is used to dynamically guide the traversal through the large search space.

Training with Reinforcement Learning involves learning from a large number of experiments and generally operates as follows. A *learning agent* interacts with an *environment*, and at each step, the agent *observes some state* in the environment, performs an *action*, and receives a *reward* from the environment. The overall goal of the learning agent is to maximize the cumulative (discounted) reward that it receives from the environment. In our case, the environment is a mobile page load setting, i.e., a combination of a device, network, and browser cache. The training process is structured as a series of *episodes*, each of which considers a single page and environment, and evaluates a running push/preload policy (starting with an empty one) that is incrementally modified to include an additional action. An action is a push/preload decision for a single object. We describe each component in more detail below.

Action/Action space: The action space lists the set of possible push/preload decisions for all objects in a page. Each action is represented as a six-tuple ($type$, $domain$, $push\_object$, $push\_ancestor$, $preload\_object$, $preload\_ancestor$). $type$ lists the action to perform (push, preload, nothing); $domain$ represents the domain whose objects to consider if the action is "push" ("preload" can consider objects from any domain); $push\_object$/$preload\_object$ and $push\_ancestor$/$preload\_ancestor$ list the object to push/preload and the object to do so with, respectively. Note that objects are identified by ID numbers here, not precise URLs, because URLs can vary on short time scales [8]; IDs are converted into URLs during inference (§3.2).

Episode: At the start of an episode, Alohamora first selects a random operating environment by picking values for the average network bandwidth, latency, and loss rate, as well as the mobile device CPU speed, and browser cache settings (time since the last load of the page, which in turn dictates cache contents [41]). Because the space of each value is continuous and thus infinitely large, we discretize each into bins that are sized according to prior work [43] and our own empirical analysis of the impact that changes to each factor have on push/preload performance. In particular, we group network bandwidth/latency/loss rate to the nearest 5Mbps/10ms/0.5%, and CPU speed to the nearest $1\times$ slowdown relative to a baseline. This lets us consider far fewer environments without sacrificing model generalizability.

In addition to the environment specification, the agent is given access to an *annotated dependency graph* for the page (Figure 5). Traditional web dependency graphs [8, 39, 43, 59] are directed acyclic graphs with a node per page object, and edges that represent initiator relationships (i.e., a parent's computation triggered the fetch of a child). We add annotations (§5.1 details how annotations are made) which list, for each object, information about its 1) size, 2) com-

putation delays, 3) content type (e.g., HTML), 4) ordering (timing) relative to both all other page objects and only those objects belonging to the same domain, 5) cache status, and 6) candidacy for push/preload. Candidacy reflects the fact that only recurring objects in a page should be considered for push/preload, in order to reduce the potential for wasting bandwidth; we determine candidacy in the same way as prior work [54], by loading the page several times and extracting a stable list of URLs. Collectively, the operating environment and annotated dependency graph represent the *observable state* that the agent can glean from the environment.

Throughout an episode, the agent selects actions according to a probability distribution over the potential space of 6-tuples (i.e., the action space described above). The probability distribution function starts as uniform, but is dynamically updated based on the agent's experiences. More formally, the agent uses a policy gradient method [33] in which it estimates the gradient of the expected total reward for each possible new action—the agent selects the action predicted to deliver the highest reward. For each new action that is added to the running push/preload policy, the updated policy is evaluated in the environment to obtain a reward that is fed back to the agent along with the *observable state*.

Each episode ends when the agent either chooses an action of *type* "nothing," repeats an action to push/preload an object that is already represented in the running policy, or selects an invalid (i.e., disallowed) action, e.g., pushing across domains or preloading an ancestor object. Regardless of which reason ends an episode, upon completion, Alohamora automatically assigns a reward of 0 to signal to the agent that the terminal policy is not one to consider. The policies learned for each episode are ultimately aggregated to generate Alohamora's overall push/preload policy generation model. We note that training can be configured to adhere to a preset bandwidth cap for pushed objects, i.e., we can end an episode if a policy that pushes an undesirably high number of bytes arises.

Reward function: Structuring the reward function requires careful thought because each action in an episode is not entirely independent. Thus, rather than simply using the page load metric of choice, we structured our reward function to take into account the relative improvement or degradation (on the metric of choice) per action, giving a boost in reward as the agent discovers a set of actions that leads to a new global (i.e., within an episode) minimum. More formally, we define the reward for the i^{th} action in an episode as:

$$R_i(P_i, P_{i-1}, P_{\text{best}}) = \begin{cases} \frac{k_1}{P_i} & P_i < P_{\text{best}} \\ \frac{k_2 P_{i-1}}{P_i} & P_i < P_{i-1} \\ \frac{-k_2 P_i}{P_{i-1}} & P_i > P_{i-1} \end{cases}$$

where P_i, P_{i-1}, and P_{best} are the raw values for the target QoE metric for the current, previous, and best-so-far policies in the episode, respectively. k_1 and k_2 are constants, where $k_1 >> k_2$. The idea is to give a positive (negative) reward proportional to an improvement (regression) in QoE. We note that the reward function is compatible with any QoE metric that denotes improved performance with lower values. We consider different reward structures in §6.5.

Implementation: Alohamora trains its models with Ray [36], using the RLLib [26] and Tune [27] libraries. Each model is a recurrent neural network that consists of 2 densely-connected layers with 256 units and the `tanh` activation function, followed by an LSTM with cell size 256. As shown in §6.5, LSTMs are helpful given the sequential nature of each episode: they prevent the agent from infinitely deferring its reward and always choosing longer policies over shorter ones. Training stops after 150 iterations, or if the standard deviation in the past 50 rewards is less than 5% of the last one (whichever comes first).

Our implementation uses the latest A3C [33] algorithm, but is compatible with others [34, 65]. As reported in prior work, A3C may incur high convergence times when network conditions or reward signals exhibit high variance during training [29]. Alohamora's training process sidesteps this in two ways. First, in addition to reducing training times, Alohamora's page load simulator eliminates noise in the observed reward signal. Second, Alohamora trains on deterministic emulated networks (including time-varying links) using Mahimahi [44], so network characteristics are unchanged within each training episode.

3.2 Online Inference

At runtime, Alohamora introduces a frontend server (or reverse proxy) that is colocated with the existing server for a domain (Figure 1); colocating ensures that end-to-end HTTPS security is preserved. All client requests first hit the Alohamora server, whose goals are to 1) collect the information required to query its policy generation model, 2) query the model, and 3) apply the suggested policy to the current load. Each origin in a page independently runs Alohamora to generate its own policy; training explores a sufficient number of policies to enable an origin to *hedge* against the set of policies that other origins may employ. We present results for partial deployment scenarios in §6.

Data collection for inference: The information required to query the policy generation model matches the *observable state* from training, i.e., network bandwidth, latency, loss rate, CPU speed, cache status, and annotated dependency graph. Alohamora collects the required network, device, and cache information through its interactions with clients, and the annotated dependency graphs directly from origin servers. Importantly, all data collection involving clients leverages existing interfaces that modern browsers already expose. In other words, *Alohamora does not require any new browser features, and instead only needs certain pre-existing features to be enabled.*

To extract network latencies, Alohamora's server analyzes the SYN/SYN-ACK time during the client's initial connec-

tion setup. Further, summaries of the client's cache are collected using either the latest cache manifest standards [46], or a server-based cookie which logs the time since the user's last load of the page [12]. We note that caching information is collected on a per-domain basis in order to preserve existing web privacy guarantees, i.e., a domain only learns about the cached objects that it owns. CPU speeds are set based on the HTTP User-Agent header that denotes the client device [37]. Lastly, average network bandwidth and loss information are collected using browser user experience reports [9].

Alohamora also requires an up-to-date dependency graph to determine the precise URLs to push/preload according to the generated policy. Alohamora relies on origin web servers to collect and share updated dependency graphs offline [30], as those servers are the first to be aware of page changes. In particular, content management systems [14, 67] support hooks that fire any time a page-altering change is pushed, e.g., for A/B testing. Alohamora adds a transparent hook to collect up-to-date dependency graphs, which requires only a lightweight (headless) load of the largely local page [40, 43].

Applying push/preload policies: Upon receiving a client request for a page, Alohamora's server queries its model to generate a push/preload policy that directly targets the current load. The resulting policy is a listing of object IDs to push/preload, and the corresponding ancestors. Alohamora then uses the latest dependency graph to translate IDs to precise URLs (according to positions in the dependency graph). Finally, to enforce the policy, the Alohamora server issues local HTTP(S) requests (mimicking client HTTP headers) to the colocated origin server, which responds with the up-to-date objects. Alohamora then applies the policy to the returned object headers throughout the rest of the page load.

4 Generalizing Across Pages

In practice, sites commonly serve thousands of different pages. Unfortunately, incorporating each page into the training process would be far too slow and resource intensive. Consequently, Alohamora faces a tricky tradeoff: train on only a few of a site's pages and achieve efficient training at the risk of omitting pages that warrant unique push/preload strategies, or train on many of a site's pages to develop generalizable policies at the expense of high training overheads.

Alohamora addresses this tradeoff by leveraging the observation that, even though sites serve thousands of different pages, those pages typically cover a *small number of page structures*, e.g., because they are automatically generated using a fixed set of templates, and thus share styles, JavaScript libraries, etc. [30, 48]. For example, news sites intuitively comprise a main home page, category home pages, and several classes of article pages. The key idea here (validated below) is that these shared structural properties typically dictate the efficacy of different push/preload strategies, and thus, we need not train on multiple pages that are structurally similar.

The primary challenge with leveraging this insight is in determining precisely which pages in a site are necessary to consider during training. Answering this implicitly requires an understanding of what pages have sufficient structural similarity from the perspective of the push/preload policies that they warrant. In other words, how should we represent and compare pages to determine structural similarity? Our goal is for representations to be coarse enough to avoid deeming all pages as structurally different (which would eliminate savings in training efficiency), but also detailed enough to capture structural differences that affect policies.

4.1 Clustering by Page Structure

We observe that the efficacy of a push/preload policy depends on the utilization of network and client device resources throughout the page load process. Building off of this, the primary determinants of resource utilization are 1) browser- and page-imposed dependencies [38, 39, 59], e.g., JavaScript execution blocking HTML parsing, and 2) the duration and overhead of different page load tasks involving the network or CPU. To capture all of these factors and identify page structures that warrant similar policies, Alohamora uses the annotated dependency graphs described in §3. Recall that the structure of these dependency graphs captures inter-object dependencies and constraints on request scheduling, while the per-object annotations characterize network and CPU overheads of fetch and execution tasks.

Given these dependency graphs (or trees) for each page that we hope to accelerate for a site, Alohamora defines the distance between two page's trees T_i and T_j as the tree edit distance between them; we use the state-of-the-art APTED algorithm [50]. The cost of inserting/deleting a node is set to 1, and the cost of each change to any part of a node's label (content type, size, execution time, etc.) is set to 0.25, i.e, label alterations are equally weighted such that changes to all labels are equivalent to a node insertion/deletion. To avoid incorporating label edits that minimally impact push/preload strategies, Alohamora deems objects that have sizes or execution times within δ% of each other as equivalent; we use $\delta = 25\%$ but find the precise value to have little impact as node insertions/deletions dominate difference values.

After computing the distances between each pair of trees, we construct a *distance matrix D* where $D_{i,j} = \text{distance}(T_i, T_j)$. With this, Alohamora can run any clustering algorithm that operates on non-Euclidean distance functions—we use agglomerative clustering [63]—to group pages that are structurally similar from a push/preload perspective. During clustering, we minimize the average distance between the pages in a cluster while permitting islands (a cluster of size 1); we sweep a range for the target number of clusters, and choose the lowest one which, if increased, does not result in a new island. From there, Alohamora only considers a single (random) page per cluster for training.

Handling page changes: Recall that origin servers track changes to their page dependency graphs and share those

graphs with Alohamora's runtime server (§3). A natural question is how to determine when a change to a page's dependency graph is substantial enough to deem Alohamora's model suboptimal (for that page) and prompt a retrain? To answer this, upon receiving a graph from an origin server, Alohamora re-clusters by computing the pair-wise distances between the new graph and graphs for all pages used in training. If the new clustering results remain stable such that the new graph falls into an existing cluster, then Alohamora needs not retrain. On the other hand, if the new page forms an island, then Alohamora will automatically trigger a re-train. During re-training, Alohamora will still use its model to service pages whose graphs have not substantially changed.

Prior work has shown that page dependency graphs remain structurally similar over long time scales (e.g., weeks), with only the precise URLs changing over short periods [8, 39, 43]. Thus, we expect retraining with Alohamora to be infrequent in practice. For example, we verified that the clustering results from Figure 4 are unchanged across 2 weeks.

4.2 Evaluations

We performed case studies on 100 randomly selected sites in the Alexa top 500 US list [3]. For each site, we ran a monkey crawler [1] that generated a list of 300 URLs by performing random interactions (e.g., clicks) starting from the site's landing page. From this set of URLs, we selected 30 pages that covered the logical clusters that we perceived for the page, e.g., articles vs. home page vs. user profile pages. For each of the 30 pages, we generated the corresponding annotated dependency graph, computed the pair-wise tree edit distances to all other pages, and performed the clustering described above. The generated clusters largely matched our high-level clustering intuition, e.g., for `The Atlantic`'s website, there exists a cluster for the home page (1), articles (21), category pages (5), and user profile pages (3).

We evaluated our clustering strategy for each site as follows. We first ran a brute force search (§2.2) to find the best push/preload policy for each of the site's considered pages. We then applied each page's best policy to all of the other pages for the site, including those in the same cluster, and those in other clusters. In each case, we measured the fraction of potential push/preload benefits that a page x's best policy achieved for another page y (as compared to the improvements delivered by y's best policy). In the event that a referenced object was missing for a page, we removed the corresponding action from the policy; this was rare as policies are based on fetch order IDs, not precise URLs.

Figure 4 lists our results for one environment; we note that the trends held for the other environments in §6.1. As shown, we find that push/preload policies are able to generalize well within a cluster, but not across clusters for a given page. In particular, at the median, policies that are generated and applied to the pages within a cluster achieve 89.6% of the potential push/preload benefits; this number drops to 36.3% for

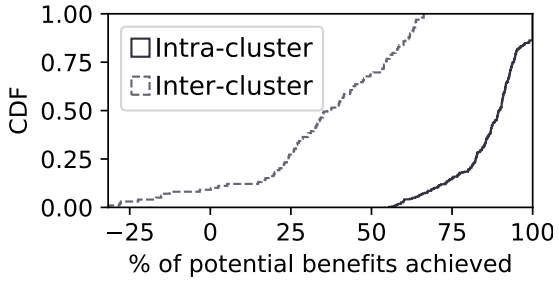


Figure 4: **Policies generalize well within (but not across) Alohamora's clusters. Results are for the {24 Mbps, 20 ms, 2× CPU slowdown, PLT} setting, and consider 100 sites, with 30 pages each. For each site, we applied each page's best policy to all other pages, and measured the % of potential benefits achieved.**

policies that are applied across clusters. §A.2 shows results for two representative pages, and also presents end-to-end results for Alohamora's policies given this clustering strategy.

5 Page Load Simulator

Even for a single page, training is impractical due to the large number of policies and environments, and the slowness of mobile page loads. To accelerate training (§3), Alohamora uses a novel page load simulator that, given an annotated dependency graph for a page, a target execution environment, and a push/preload policy as input, outputs an estimated QoE (e.g., PLT, SI) value. Unlike prior simulators (§A.1), Alohamora's is able to faithfully predict load performance (with any policy) across metrics and environments, without requiring costly profiles [68] or emulation [60] per environment—this is critical for Alohamora's training as loading pages in each environment would forego most simulation speedups. We will start by describing our simulator's operation in the context of cold cache loads, no push/preload, and PLT, and then relax those assumptions in §5.4 and §A.1. We note that Alohamora's simulator focuses on HTTP/2 page loads.

5.1 Collecting Simulator Inputs

The first step in the simulation process is to profile a load of the target page to extract information characterizing properties dictated by page composition [39] or browser dependencies [38, 59]. These properties do not describe the operating environment (which we will simulate), but instead dictate how page load tasks should share the simulated resources.

To extract such information, Alohamora records the target page with a record-and-replay tool [44], and replays the page over an unshaped local network with desktop-level CPU resources. During replay, Alohamora extracts an annotated dependency graph (Figure 5) that matches the ones used in §3 and §4.1. In particular, the graph structure captures the inter-object ordering and dependency constraints, and we add additional annotations that characterize each object's size, execution time, content type, etc. To aid simulation, we further break down an object's network and compute delays into:

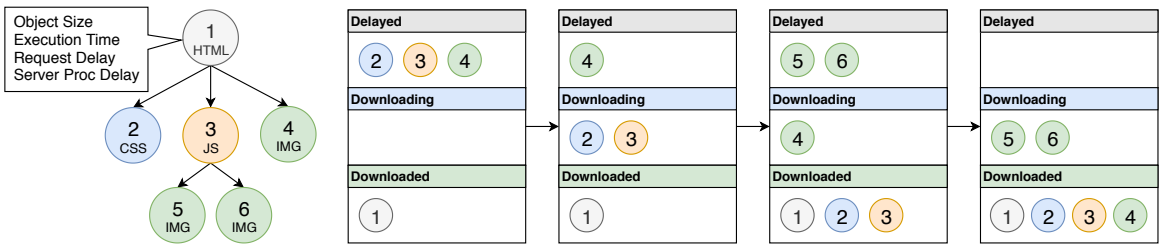


Figure 5: **Operation of Alohamora's page load simulator. The simulator operates in steps, as objects flow through these three queues, incurring blocking (e.g., connection setup, inter-object dependencies), network, and compute delays, respectively. Once an object is executed, its children are added to the top (as *delayed*) to simulate the browser discovering those dependencies.**

- **execution time:** time spent parsing, executing, or rendering the object with the well-provisioned CPU; this does not include the time to execute any referenced objects.
- **request delay:** the amount of time between when the object's parent has finished downloading, and when the object's request is issued; this embeds the parsing/execution delays of the parent, as well as any synchronous processing delays for objects referenced earlier in the parent's execution, e.g., a blocking external `<script>`.
- **server-processing delay:** server-side delay in generating and serving the response; we extract this information directly from web record-and-replay frameworks [30, 44].

In addition to this dependency graph, Alohamora's simulator also takes as input an environmental specification, listing the average network bandwidth, latency, and loss rate (Mbps, ms, %), device CPU speed (slowdown compared to profiling CPU speed), and browser cache contents.

5.2 Simulating the Execution Environment

In order to enforce the specified network and CPU values on all page load tasks, Alohamora's simulator uses a new **Request Queue** abstraction. Here, we describe how the Request Queue operates on objects passed into it; we will then describe how objects get added to the Request Queue.

At any time, the Request Queue keeps track of three types of objects using three subqueues: *delayed*, *downloading*, and *downloaded*. *Delayed* objects are those that have been discovered by the browser, but whose downloads are currently blocked, e.g., due to connection setup delays or the object's *request delay*; *downloading* objects are currently being fetched over the network; *downloaded* objects have been fetched and are currently being evaluated (or awaiting evaluation). At a high level, the Request Queue operates in steps, whereby objects flow through these queues, and once executed, children are added to the top (as *delayed*) to simulate the browser discovering those dependencies. In order to determine how long an object stays in each queue, the Request Queue models the interaction between the browser and environment, with respect to network and CPU usage.

Enforcing latency/loss overheads: In order to compute the number of round trips required to download an object, the Request Queue considers two factors. First, if the object is the first to be downloaded from a given domain, the Request Queue adds 2 RTTs to account for the TLS handshake that HTTP/2 mandates. Second, the Request Queue estimates the number of round trips required for the TCP-level data transfer by (approximately) keeping track of TCP window state for each connection (assuming cubic) and assuming that concurrent objects fairly share the window. More specifically, it assumes an initial congestion window of 10 [22], additively increases the window as bytes are downloaded, and halves the window on each idle RTO (200 ms) or probabilistic packet loss. Note that these round trip counts are computed when an object is added to the Request Queue, and are thus approximate since currently downloading objects may complete prior to the new object moving to *downloading*.

Enforcing bandwidth overheads: Across all concurrently *downloading* objects, the Request Queue must enforce an appropriate split of the specified network bandwidth. The simulator treats the bandwidth specification (either average bandwidth or a packet delivery trace [44]) as characterizing the access link, which is commonly the bottleneck in wireless networks [66] and is shared by all origins' connections. By default, the Request Queue assumes that outstanding requests fairly share the available bandwidth, thereby disregarding discrepancies in cross-connection window state.

Enforcing CPU overheads: The Request Queue modulates the *execution delay* and *request delay* for each object by multiplying by the magnitude of the CPU slowdown factor. The simulator ignores CPU core counts, and instead focuses on clock speeds, which have been shown to be the main factor affecting browser performance [10]. To support parallel iframe execution, the Request Queue subtracts out execution times from concurrently *delayed* objects across frames.

Request Queue operation: The Request Queue proceeds in discrete "steps". In each, the Request Queue inspects the lists of *downloaded*, *downloading*, and *delayed* objects, and finds the object(s) that are scheduled to either finish execution first, finish downloading first (fewest bytes remaining), or transition to *downloading* soonest, respectively. Each step is clocked by the duration t until those object events complete. After computing t, the Request Queue will subtract t from the *execution delays* of all *downloaded* objects, subtract the number of bytes that can be downloaded in t from all cur-

rently *downloading* objects, and subtract t from the blocking delays for all currently *delayed* objects. It will then move all *delayed* objects whose blocking delays have expired to the *downloading* queue, and mark all objects that complete *downloading* as *downloaded*. We discuss how *downloaded* objects affect subsequent resource discovery next.

5.3 Simulating Page Loads

Starting from the root node in the dependency graph (i.e., the top-level HTML), each time an object is marked as *downloaded* by the Request Queue, the simulator immediately adds all of that object's direct children as *delayed* to the Request Queue, simulating the browser's discovery of those objects. In other words, each child of the completed object is scheduled in a one-step look-ahead process, resulting in a dependency graph traversal that is breadth-first across each object's children, but not necessarily across siblings with different parents (Figure 5). Note that, after its children are added to the Request Queue, the parent object remains in the *downloaded* queue until its *execution delay* expires; in parallel, each child incurs its own *request delay* which characterizes the offset in the parent's execution until it is discovered.

This simple approach closely mimics the browser graph traversal strategy [39, 59], but with one issue: execution dependencies between an object's children. For instance, consider a simple scenario in which the top-level HTML includes two adjacent HTML `<script>` tags that reference files $S1$ and $S2$, both of which have children. Because browsers are unaware of the potential state dependencies between these two JavaScript files, upon discovering the first `<script>` tag, HTML parsing would halt and trigger a synchronous (i.e., blocking) fetch and execution of $S1$ [39]. This has several implications on dependency graph traversal, which Alohamora's simulator must account for:

- during a real load, the children of a parent may not be scheduled in a single burst. The simulator accounts for this by including an object's *request delay* (which accounts for inter-children blocking delays) in the duration that the Request Queue marks it as *delayed* (§5.2).
- even with the enforced *request delay*, it is possible for the Request Queue to mark $S2$ as *downloaded* before $S1$, e.g., if $S2$ is far smaller and the simulated network is bandwidth-constrained. This could result in cascading discrepancies in graph traversal since $S1$'s children should be handled before $S2$'s. To handle this, the simulator also exposes the Request Queue to IDs listing the object fetch orders logged in the profiled load. These IDs inherently follow the order in which browsers require, or are blocked on, specific objects. With this information, the Request Queue treats *downloaded* objects as a priority queue, signaling object completion to the graph traversal component only once the next required object (i.e., the lowest incomplete ID) is complete. Asynchronously-fetched objects are returned after their closest synchronous neighbors.

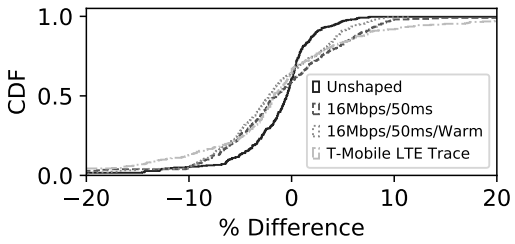


Figure 6: **Faithfulness of the Alohamora simulator's predicted PLT compared to measurements from a real browser.**

We note that, despite these strategies, the simulator's dependency graph traversal still faces potential inaccuracy in the fact that objects involving a blocking dependency, such as $S1$ and $S2$ in the above example, may download concurrently and share network resources. However, the simulator bounds the cascading effects of these inaccuracies on the page load process by ensuring that the ordering of downstream object discovery faithfully mimics that of a real browser.

Measuring PLT: As *downloaded* objects complete execution in the Request Queue, they are marked with a completion time relative to the start of the page load. PLT is the maximum object completion time [42]. In §A.1, we discuss how other metrics such as Speed Index are measured.

5.4 Simulating Push/Preload Policies

To support push/preload, when an object is being added to the Request Queue, the simulator also schedules the corresponding objects to push and preload along with that object (as per the input policy). The objects added for push/preload largely share the blocking delays of the ancestor since push/preload objects cannot begin downloading until the ancestor does. In particular, the Request Queue imposes the ancestor's *request delay*, but alters the remaining delays in two ways: 1) their server-side processing delays are preserved (and not adopted from the ancestor), and 2) preload objects incur an additional network RTT to account for the download of the ancestor's response HTTP headers (0.5 RTT) and transmission of the preloaded object's request (0.5 RTT).

Once scheduled, the key challenge is in determining how pushed/preloaded objects affect the delays from the profiled load; this delta could be positive or negative due to, e.g., bandwidth contention. To understand this, once the simulator hits a pushed/preloaded object, it determines how the object's download progress compares (or will compare) to the case when the object was not pushed/preloaded. This is done by simulating the load without that object being pushed/preloaded, and comparing the resulting delays. Note that, if the pushed/preloaded object is blocking, delays for downstream siblings are edited to reflect the observed deltas.

5.5 Evaluations

We evaluated our simulator by comparing to a real browser on two metrics: fidelity in predicted performance and overall runtime. We follow the same setup as described in §6.1.

	Median	95th Percentile
Alohamora's simulator	4.7	22
Unshaped	1347	3815
24Mbps/20ms/2x CPU	5936	16683
12Mbps/60ms/4x CPU	9631	27765

Table 1: **Per-page runtimes (ms) of Alohamora's simulator (top row) and a real browser in different execution environments.**

Fidelity: Figure 6 shows that Alohamora's simulator reports highly faithful load times compared to a real browser. For example, in an environment with no network or CPU shaping and a cold browser cache, the simulator's reported load times were within 0.4% and 4.3% of the real browser, at the median and 95th percentile, respectively. Median discrepancies marginally increase to 1.4%, 1.7%, and 2.2% as fixed-rate (16 Mbps, 50 ms link) and time-varying (T-Mobile LTE) network shaping, and caching are incrementally added; we note that the errors for all other tested environments are within 4% of these numbers. Further, the low error rates persist when evaluating push/preload policies (§A.1).

Runtime: As shown in Table 1, Alohamora's simulator evaluates page loads 3-4 orders of magnitude faster than real browsers, with the discrepancies growing as the target environment becomes more resource-constrained. §A.1 describes how simulation times vary with policy length. For context, these runtime savings enable Alohamora to *reduce the training time for a page from 10s of days to just 10s of minutes*.

6 Evaluation

6.1 Methodology

To create a reproducible test environment and cover a wide range of environments, our evaluation mainly involves emulation using the Mahimahi record-and-replay tool [44]; we present real-world experiments in §6.4. Our main corpus comprised the Alexa top 500 US landing pages [3], but we also used non-landing and less popular pages in §A.2. We recorded versions of each page at multiple times to mimic different warm cache scenarios: back to back loads, and loads separated by 4, 12, and 24 hours. Mobile-optimized (including AMP [21]) pages were used when available. Experiments used Google Chrome for Android (v72).

Our emulation evaluation considered a broad range of network bandwidths (6-48 Mbps, as well as Verizon and AT&T LTE traces [44]), latencies (0-100 ms), loss rates (0.5-5%), and client device conditions (CPU slowdowns of 1-4×, relative to a desktop with an Intel Xeon Gold 5220 CPU @ 2.20GHz). Network emulation was performed using Mahimahi [44], and CPU constraints were enforced using Chrome's Devtools Protocol [13]. Unless otherwise noted, Alohamora generated a single policy generation model per page that covered the aforementioned conditions; results for Alohamora's cross-page models based on clustering (§4) are shown in §6.6. Further, in accordance with §3, dependency graphs for inference were made just prior to the experiments.

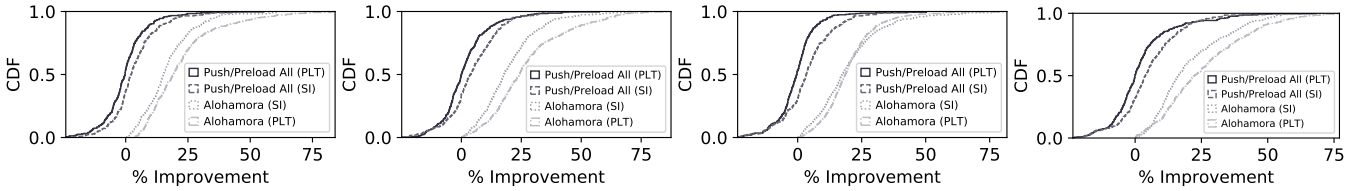
We compared Alohamora to default page loads (i.e., no push/preload) and two standard push/preload strategies: 1) the *push/preload all* strategy where, on the first incoming request, each origin pushes all static resources that it owns, and preloads all referenced static third-party resources, and 2) the *push/preload all JavaScript* strategy which operates in the same manner but only considers JavaScript objects that (unlike images) may trigger subsequent object fetches. With both strategies, push/preload order matches the order in which objects are referenced by a page. Our analysis, described in §A.2, revealed that *push/preload all* consistently delivers larger speedups than *push/preload all JavaScript*. Thus, for brevity, we only present results comparing Alohamora with *push/preload all*.

Our evaluation considers two performance metrics: page load time (PLT) measured as the time between the `navigationStart` and `onload` events, and Speed Index (SI) (measured with pwmetrics [24]) which captures the time needed to fully render the initial viewport. Due to space constraints, we present results for select settings, but note that presented trends persist in all tested scenarios. For all results, domains make push/preload decisions *independently* with Alohamora, and objects can only be pushed within a given domain, e.g., `google.com` cannot push an image belonging to `g.static` (which Google owns).

6.2 Page Load Speedups

Cold cache: Figure 7 illustrates Alohamora's ability to accelerate cold cache page loads across four representative settings. For example, in a {24 Mbps, 20 ms, 1× CPU slowdown, 0% loss} environment, median (95th percentile) PLT improvements with Alohamora were 24% (61%); the push/preload all strategy achieved only -0.2% (22%) improvements. Alohamora's benefits persist as network and CPU conditions change, although the generated policies vary: benefits are 19% (45%) when conditions degrade to {18 Mbps, 60 ms, 4× CPU slowdown, 0% loss}, 22% (57%) when 1% loss is introduced, and 14% (60%) over a time-varying Verizon LTE trace (not shown). Figure 7 also shows that Alohamora provides substantial SI benefits, ranging from 15-19% and 36-48% at the median and 95th percentile. Importantly, across all settings, Alohamora's push/preload policies *never degraded performance* compared to a default page load. This is in stark contrast to the static push/preload all policy, which slowed down 40% of pages by up to 22%.

Warm cache: Figure 8 shows that, across a wide range of warm cache browsing scenarios, Alohamora accelerates page loads compared to both a default page load and a static push/preload all strategy. For instance, for back-to-back (i.e., perfectly warm-cache) page loads, median PLT improvements are 0.9 s and 0.4 s for Alohamora and the push/preload all strategy, respectively. These relative improvements persist (12-18%) as the time between page loads increases.



(a) **12 Mbps, 100 ms, 2×, 0%** (b) **24 Mbps, 20 ms, 1×, 0%** (c) **18 Mbps, 60 ms, 4×, 0%** (d) **18 Mbps, 60 ms, 4×, 1%**

Figure 7: **Load time (PLT and SI) improvements over a default page load for a static push/preload all strategy, and Alohamora. Environments are listed as {bandwidth, latency, CPU slowdown, loss rate}. Results used cold browser caches.**

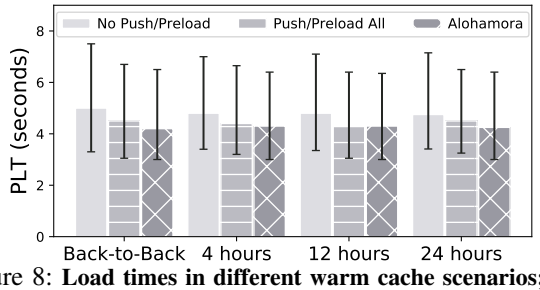
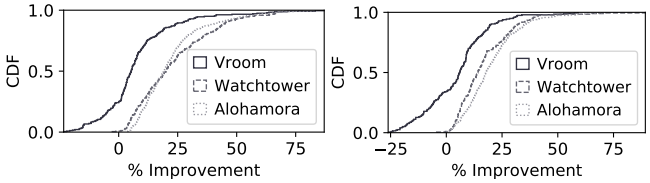


Figure 8: **Load times in different warm cache scenarios; "No push/preload" is a default page load. Bars represent medians, with errors bars spanning 25-75th percentiles. Results are for the {12 Mbps, 100 ms, 2×, 0%} setting.**



(a) 12Mbps, 100ms, 2×, 0%, PLT (b) 24 Mbps, 20 ms, 1×, 0%, SI

Figure 9: **Comparison with Vroom [54] and WatchTower [43].**

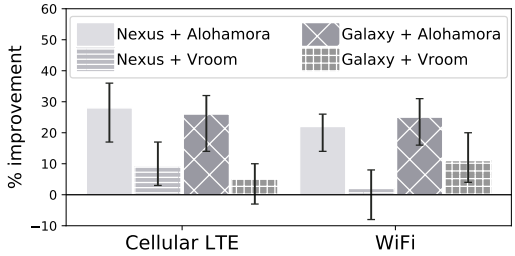


Figure 10: **PLT improvements over a default page load with Alohamora and Vroom, in the wild. Results used cold caches.**

6.3 Comparison to State-of-the-Art

We compared Alohamora with two recent mobile web accelerators, Vroom [54] and WatchTower [43]. Vroom improves upon the push/preload all policy by using a client-side scheduler to integrate priorities into the ordering of pushed/preloaded objects. In contrast, WatchTower selectively uses proxies (per origin) that fetch objects on behalf of clients using fast wired networks. Client-origin-proxy latencies were set as if proxies were run on Amazon EC2 in California, and WatchTower ran in HTTPS-sharding mode.

As shown in Figure 9, Alohamora outperforms Vroom on both PLT and SI. For example, in a {12 Mbps, 100 ms, 2× CPU slowdown, 0%, PLT} environment, benefits with Alohamora are 3.6× and 1.4× higher than Vroom's at the median and 95th percentile, respectively. The main reason for this discrepancy is that, even though Vroom adds dynamism to push/preload in the form of priority-based scheduling, Vroom remains too constrained to adapt to diverse execution environments. In particular, the set of objects to push/preload are static and match the push/preload all approach. This is partly evidenced by the fact that Vroom still harms a large fraction of page loads, e.g., 34% in the {24 Mbps, 20 ms, 1× CPU slowdown, SI} setting. In contrast, Alohamora can vary all aspects of the push/preload policy (objects, orderings, etc.) to best cater to the target setting, and *never harm performance*. Figure 9 also shows that Alohamora marginally outperforms WatchTower (0.9-1.7× more median benefits) *without* requiring per-origin proxy servers.

6.4 Real-World Experiments

We also evaluated Alohamora in the wild, using the same 500-page corpus from §6.1, live Verizon LTE and residential WiFi networks, and 2 mobile phones: a Nexus 6 (Android Nougat; 2.7 GHz quad core processor; 3 GB RAM) and a Galaxy Note 8 (Android Oreo, 2.4 GHz octa core; 6 GB RAM). To apply Alohamora's policies without relying on origin web server modifications, our setup uses an NGINX reverse proxy server [45]. The proxy was run on a c4.large Amazon EC2 instance in California, which had a median latency of 11 ms to the origin web servers in our corpus.[1]

Immediately prior to the experiment, the proxy downloaded the dependency graph for each page, and all of the static objects in the graph that are candidates for pushing. The proxy also housed Alohamora's learned model for each page. At runtime, all requests from the mobile device were forwarded to the proxy via DNS rules; even with a single proxy, browsers still opened a separate connection per origin server since connection setup decisions are based on domain name (not IP address). Upon receiving the first request per origin in a page, the proxy generated and applied a policy (for that origin), pushing cached resources and rewriting HTTP headers to reflect preload decisions. The proxy could also apply Vroom's policies or forward requests to origin servers.

As shown in Figure 10, median PLT improvements were 2.3-11× higher with Alohamora than Vroom; SI results followed the same trend, but were elided for space. Figure 10

[1]These proxy-to-origin latencies present a pessimistic setting, since Alohamora is designed to run directly on origin web servers.

	Reward	LSTM	BW	CPU	Latency	Loss
C1	66 (97)	61 (93)	49 (93)	48 (90)	57 (94)	59 (91)
C2	69 (97)	65 (95)	58 (95)	54 (94)	59 (92)	62 (92)

Table 2: **Impact of removing features/properties in Alohamora's model. Results are reported as median (95th percentile) percentage of potential PLT improvements (compared to Alohamora's full model). "Reward" considers the intuitive $-PLT$ reward function. C1 and C2 are the {12 Mbps, 100 ms, 2×, 0%} and {24 Mbps, 20 ms, 1×, 1%} settings, respectively.**

also illustrates Alohamora's ability to properly adapt to conditions in the wild: whereas Vroom harms performance for up to 43% of pages, Alohamora *always* sped up loads.

6.5 Understanding Alohamora's Benefits

Ablation study: To understand the relative impact of each of Alohamora's features and model properties, we performed an ablation study (Table 2). Our results reveal that bandwidth, latency, CPU, and loss information all play significant roles in Alohamora's ability to generate performant push/preload policies, with the removal of CPU inputs resulting in the largest median degradations (46-52%). Our results also highlight the importance of Alohamora's reward function and incorporation of LSTM. For instance, (intuitively) setting the reward to $-PLT$ leads to performance degradations of around 30% because it becomes easy for the agent to artificially inflate the observed reward by selecting policies with fewer actions, i.e., the earlier policies in an episode will be favored as the cumulative reward will be lower. Removing LSTM, on the other hand, led to degradations of ∼35%, largely due to the lack of a discount factor that guides the agent to avoid unnecessarily favoring longer policies (§3).

Alohamora's policies: To understand the learned insights behind Alohamora's benefits, we analyzed its generated push/preload policies. Admittedly, we observe that policy composition and the mix between push/preload varied dramatically across pages and resource settings; indeed, subtle interactions between these properties were a primary motivator for Alohamora's machine learning-based approach. However, we note the following common principles:

- In lower bandwidth settings, Alohamora either 1) reduced the policy length or cut data-intensive objects, or more commonly, 2) spread the same set of pushed/preloaded objects out across a larger set of parents in order to stagger downloads and reduce bandwidth contention.
- With slower CPUs, Alohamora's policies are careful to only push objects whose bytes could be downloaded until the next blocking JavaScript file is required; the goal is to prevent downstream CPU tasks from blocking on the network. In these cases, Alohamora's policies preloaded additional resources with the goal of having their downloads start (after the 0.5 RTT to contact the server) only after the next blocking resource was downloaded. In essence, the idea is to perfectly interleave downloads of non-blocking resources with the execution of blocking resources.
- For image-heavy sites (e.g., `pinterest.com`), Alohamora commonly excluded JavaScript/CSS files from its policies, and instead pushed/preloaded images that are rendered towards the top of the viewport, particularly in high-bandwidth settings or when SI is the target metric. The reason is that these pages have flat (not deep) dependency graphs, so blocking JavaScript files do not trigger cascaded serial network fetches; instead, image downloads have a larger blocking impact on load times.

We leave a more detailed analysis of Alohamora's generated policies, and an exploration into whether those policies could be converted into fixed general heuristics, to future work.

Unnecessary data usage: A well-documented risk with HTTP/2 push is in having servers push objects that are not needed by or already cached at the client browser [43, 54]. Alohamora avoids this issue in two ways: browser cache contents are explicitly considered during policy generation, and only resources that consistently appear in a page are considered for push/preload (§3.1). Consequently, we observe that Alohamora's policies do not waste any bandwidth, i.e., all pushed/preloaded objects are used in the targeted page load.

Training times: Training an Alohamora model for the median page in our corpus required 76 iterations and took a total of 19.4 minutes to reach convergence using an Amazon EC2 `c5.18xlarge` instance. This translates to a monetary cost of $0.62. We note that training costs are incurred offline and infrequently: Alohamora must retrain only when a new or modified page falls outside of the previous cross-page clusters, which we observe occurs on the order of weeks (§4.1).

Inference times: Alohamora's policy generation adds negligible delays to overall load times: median (95th percentile) inference times are 11 ms (40 ms), respectively.

6.6 Additional Results

We briefly summarize our remaining experiments here due to space constraints, and defer details to the §A.2.

Incremental deployment: We ran experiments to understand how benefits vary with different adoption rates. We found that benefits (unsurprisingly) increase as more domains adopt Alohamora, but simply having the top-level origin can achieve 56% of the potential median benefits.

Cross-page clustering: We performed an end-to-end evaluation of Alohamora's clustering strategy (§4) by training a model for each of the 100 sites in Figure 4, using only a single page per cluster. These models achieve 85-90% of the improvements achieved when training on pages individually.

Other pages, input errors, and energy savings: Alohamora's benefits persist (and in fact, increase) for interior pages and less popular sites, are robust to errors in network or device measurements, and yield per-page mobile device energy reductions of 16-23%.

7 Related Work

We discuss the most closely related work here, and present additional related work in §A.3.

Server push systems: Numerous studies have explored the performance of HTTP/2 (formerly SPDY), both with and without server push and preload [16, 20, 53, 54, 60, 70]. Like us, these works have found mixed performance benefits due to the subtle relationships between HTTP/2 and network characteristics, page composition, and TCP semantics. However, these prior efforts have all investigated and promoted static policies and configuration guidelines. In contrast, Alohamora leverages a data-driven approach to dynamically tune push/preload policies by explicitly factoring in both page composition and the target execution environment.

Mobile-optimized pages: Certain systems, most notably Prophecy [40], automatically rewrite web pages and return post-processed versions of objects to clients that reduce client compute and network costs. Unlike Prophecy, Alohamora does not alter page content, which has proven to be error-prone in practice [2]. Further, Alohamora can accelerate Prophecy pages which require at least one HTML file per frame, and unmodified image and style files—these are the static files which Alohamora targets for push/preload.

Proxy-based accelerators: Compression proxies [2, 47, 55, 58] compress objects in-flight between clients and servers, while remote dependency resolution proxies [43, 44, 56, 57] perform certain object fetches and computations on behalf of clients. Though performant, such acceleration proxies violate the end-to-end security guarantees of HTTPS. WatchTower [43] addresses this dilemma, but at a significant deployment cost, as each origin in a page must operate its own proxy. Alohamora avoids such security concerns by relying only on end-to-end HTTP/2 optimizations.

Dependency-aware scheduling: Klotski [8] analyzes pages offline to identify high-priority objects, and uses knowledge of network bandwidth and page structure to stream them to clients before they are needed. Klotski's dynamic prioritization hinges on global knowledge of object fetches, which proxies provide at the cost of security; in contrast, Alohamora origins operate independently and hedge against the decisions that other origins may make. Polaris [39] uses a client-side scheduler that reorders requests to minimize serial round trips without violating dependencies. However, unlike Alohamora, Polaris relies on clients to discover page resources, and thus cannot eliminate certain serial fetches.

8 Conclusion

Configuring HTTP/2 push/preload policies has proven challenging, as benefits depend on complex interactions between dynamic page, network, device, and browser properties. This paper presents Alohamora, a mobile web optimization system that dynamically generates HTTP/2 push/preload policies using Reinforcement Learning. To ensure practicality, Alohamora introduces novel techniques that drastically reduce the number of pages to consider for training, and the cost of training any one page—these benefits come without a drop in model generalizability. Across a broad range of settings, we find that Alohamora outperforms default loads and recent push systems by 19-61% and 3.6-4×, respectively.

Acknowledgements. We thank Harsha Madhyastha, Vaspol Ruamviboonsuk, and Anirudh Sivaraman for their valuable feedback on earlier drafts of this paper. We also thank our shepherd, Aruna Balasubramanian, and the anonymous NSDI reviewers for their constructive comments. This work was supported in part by NSF grant CNS-1943621.

References

[1] SeleniumHQ Browser Automation. https://selenium.dev/, 2019.

[2] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI '15. USENIX, 2015.

[3] Alexa. Top Sites in the United States. http://www.alexa.com/topsites/countries/US, 2018.

[4] D. An. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 2018.

[5] M. Belshe, M. Thomson, and R. Peon. Hypertext transfer protocol version 2 (HTTP/2). 2015.

[6] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-perceived Quality into Web Server Design. World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking, 2000.

[7] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. CHI, The Hague, The Netherlands, 2000. ACM.

[8] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[9] G. Chrome. Chrome User Experience Report.

[10] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of Device Performance on Mobile Internet QoE. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, pages 1–7. ACM, 2018.

[11] A. Davies. HTTP/2: Discover the Performance Impacts of Effective Prioritization. https://developer.akamai.

com/blog/2019/01/31/http2-discover-performance-impacts-effective-prioritization, 2019.

[12] DeNA Co., Ltd. H2O Cache-Aware Push. https://h2o.examp1e.net/configure/http2_directives.html, 2019.

[13] G. Developers. Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/.

[14] Drupal. Drupal - Open Source CMS. https://www.drupal.org/, 2019.

[15] E. Enge. MOBILE VS. DESKTOP USAGE IN 2019. https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study, 2019.

[16] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? *IEEE/ACM Trans. Netw.*, 23(6):2010–2023, Dec. 2015.

[17] D. Etherington. Mobile internet use passes desktop for the first time, study finds. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/, 2016.

[18] T. Everts and T. Kadlec. WPO stats. https://wpostats.com/, 2019.

[19] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems, 2004.

[20] U. Goel, M. Steiner, M. P. Wittie, M. Flack, and S. Ludin. Http/2 performance in cellular networks: Poster. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 433–434. ACM, 2016.

[21] Google. Accelerated Mobile Pages Project – AMP. https://www.ampproject.org/.

[22] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[23] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[24] P. Irish. pwmetrics: Progressive web metrics. https://github.com/paulirish/pwmetrics, 2019.

[25] B. Jun, F. E. Bustamante, S. Y. Whang, and Z. S. Bischof. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2019.

[26] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

[27] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[28] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[29] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *International Conference on Learning Representations*, 2019.

[30] S. Mardani, M. Singh, and R. Netravali. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2020. USENIX Association.

[31] P. Meenan. HTTP/2 Prioritization. https://calendar.perfplanet.com/2018/http2-prioritization/, 2018.

[32] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1928–1937. JMLR.org, 2016.

[33] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[35] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2018.

[36] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577. USENIX Association, 2018.

[37] U. Naseer and T. Benson. Configtron: Tackling network diversity with heterogeneous configurations, 2019.

[38] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[39] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th*

USENIX Conference on Networked Systems Design and Implementation, NSDI, Berkeley, CA, USA, 2016. USENIX Association.

[40] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[41] R. Netravali and J. 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*, HotMobile '18, pages 63–68. ACM, 2018.

[42] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Renton, WA, USA, 2018. USENIX Association.

[43] R. Netravali, A. Sivaraman, J. Mickens, and H. 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*, MobiSys '19, pages 430–443. ACM, 2019.

[44] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[45] NGINX. NGINX Reverse Proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/, 2019.

[46] K. Oku and Y. Weiss. Cache Digests for HTTP/2. https://httpwg.org/http-extensions/cache-digest.html, 2019.

[47] Opera. Opera Turbo. http://www.opera.com/turbo, 2018.

[48] Optimizely. Content Management System. https://www.optimizely.com/optimization-glossary/content-management-system/, 2019.

[49] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[50] M. Pawlik and N. Augsten. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, Mar. 2015.

[51] C. Petrov. 52 Mobile vs. Desktop Usage Statistics For 2019 [Mobile's Overtaking!]. https://techjury.net/stats-about/mobile-vs-desktop-usage/, 2019.

[52] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. give in to procrastination and stop prefetching.

[53] S. Rosen, B. Han, S. Hao, Z. M. Mao, and F. 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*, WWW. International World Wide Web Conferences Steering Committee, 2017.

[54] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. 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. ACM, 2017.

[55] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[56] A. Sivakumar, C. Jiang, S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[57] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. 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 '14, pages 325–336, New York, NY, USA, 2014. ACM.

[58] J. Volpe. Nokia Xpress brings cloud-based compression to the Lumia line. Engadget. https://www.engadget.com/2012/10/03/nokia-xpress-brings-cloud-based-compression-to-the-lumia-line/, October 3, 2012.

[59] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[60] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[61] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[62] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12. ACM, 2012.

[63] J. H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[64] M. Wijnants, R. Marx, P. Quax, and W. Lamotte. HTTP/2 Prioritization and Its Impact on Web Performance. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 1755–1764, 2018.

[65] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[66] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[67] WordPress. Blog Tool, Publishing Platform, and CMS – WordPress. https://wordpress.org/, 2019.

[68] K. Zarifis, M. Holland, M. Jain, E. Katz-Bassett, and R. Govindan. Modeling HTTP/2 speed from HTTP/1 traces. In *International Conference on Passive and Active Network Measurement*, pages 233–247. Springer, 2016.

[69] T. Zimmermann and O. Hohlfeld. Skip to the article Adoption, performance, and human perception of HTTP/2 Server Push. https://blog.apnic.net/2018/04/26/adoption-performance-and-human-perception-of-http-2-server-push/, 2018.

[70] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle. Is the Web ready for HTTP/2 Server Push? In *Proceedings of the 14th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2018.

Length	0	1-9	10-19	20-29	30-39
Runtime	4.7 (22)	12 (73)	45 (189)	105 (348)	172 (546)

Table 3: **Median (95th percentile) simulator runtimes in milliseconds with varying push/preload policy length.**

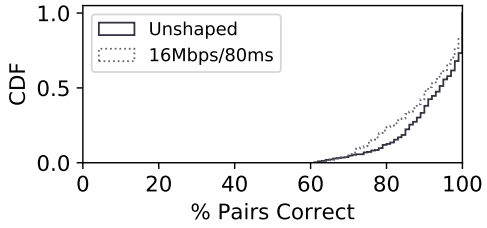


Figure 11: **Alohamora's simulator is able to correctly compare push policy pairs (in terms of relative performance).**

A Appendix

A.1 Additional Simulator Details

Additional performance metrics: We extended Alohamora's simulator to return the Speed Index and above-the-fold time [42], which is the time-instant version of Speed Index (§6.1). For this, during profiling, the simulator determines the positional information for each page component. Note that this information is dictated by page content, and can be parsed in relation to the target viewport size, i.e., we can collect positional coordinates for each page component during the profiling load, and then determine the visible content for any given viewport size [42]. With this information, the simulator identifies the set of page objects that affect the visual aspects of the target browser viewport, and characterizes performance as the time when the last node in the collected set completes its load. The simulator is also amenable to other performance metrics. For instance, to evaluate Ready Index [42], the profiling step must measure the fraction of the viewport that is visually or functionally affected by each object's execution; performance would be progressively tracked as the weighted average between time and each object's fraction.

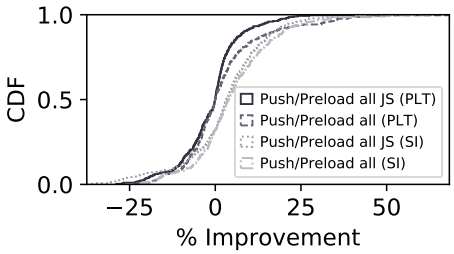
Warm cache page loads: In order to handle warm-cache browsing scenarios, the simulator takes an additional input: the list of resources that it should consider as cached, which can be computed by analyzing HTTP headers according to a desired warm cache timing, i.e., the time between the cold and warm cache page load [41]. The simulator then operates as normal, but sets the network RTTs required to fetch a cached resource, and the bytes that must be downloaded, to 0; *request delays* for downstream children of blocking resource are also updated.

Simulator speed vs. push/preload policy length: Table 3 shows that the simulator's runtime does steadily increase as the length of the push/preload policy under test grows. The reason is that Alohamora's approach to handling push/preload policies requires re-simulations of the page a number of times that is quadratic with the policy length. We note, however, that the resulting runtimes are still several orders of magnitude lower than default browsers, and Alohamora rarely requires investigation of policies longer than 20 objects (§6).

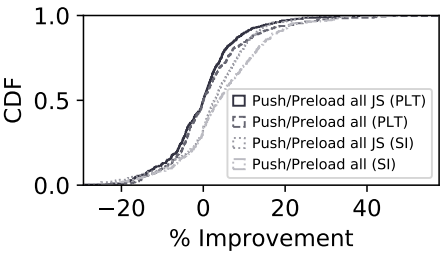
Push/preload fidelity results: §5 presented results showing the low error rates that Alohamora's simulator achieves for page loads that do not use push/preload policies. With respect to push/preload policies, the key property required for Alohamora's training is to be able to determine which of two policies results in superior performance. To evaluate the simulator's faithfulness for this, we generated 20 random push/preload policies for each page in our corpus. For each page, we counted the fraction of policy pairs for which the simulator correctly predicted the relative performance (correctness was defined by a real browser). As shown in Figure 11, the simulator correctly reported the relative comparisons across pairs 90% of the time.

Comparison to prior simulators. Several recent works have proposed web page load simulators and emulators. Here we briefly describe these prior approaches, explain their limitations for Alohamora's training scenario, and contrast them with the operation of Alohamora's simulator.

- EPLOAD [60] controls the variability in the page load process (for reproducible measurements) by profiling a page load and recording fine-grained delays between browser compute tasks (including dependencies captured by WProf [59]). EPLOAD then replays the page load process by replaying those blocking delays (via injected sleeps), but making fetches over a live (controlled) network. Thus, EPLOAD *emulates* the page load process, running live network tasks and forcing compute delays to match those from the profiled load. In contrast, Alohamora *simulates* the entire page load process, by respecting the invariant dependencies enforced by a browser and page content, as well as by modeling the interactions between the browser and underlying environment. This difference is critical to Alohamora's simulation goals: simulation enables Alohamora to evaluate push/preload policies in a few milliseconds (rather than 10s of seconds); EPLOAD's emulation approach does not shrink load times, and instead focuses on fine-grained reproducibility. Consequently, EPLOAD would not be able to accelerate training with Alohamora. Beyond this fundamental difference, Alohamora's simulator also is able to evaluate HTTP/2 push/preload policies, simulate a variety of environmental factors (CPU speeds, etc.), and evaluate performance on multiple performance metrics (e.g., Speed Index)—EPLOAD lacks these features.
- RT-H2 [68] uses profiles of HTTP/1.1 page loads to estimate the predicted performance changes for the scenario when those profiled loads were converted to using HTTP/2. The system's page load conversion model considers how HTTP/2 features (e.g., request multiplexing) affect the ordering of different page load tasks, as well as



(a) 18 Mbps, 60 ms, 4×, 1%



(b) 24 Mbps, 20 ms, 1×, 0%

Figure 12: **Performance comparisons between two existing (standard) push/preload strategies: push/preload all and push/preload all JavaScript (JS). Results are relative to default page loads (i.e., no push/preload).**

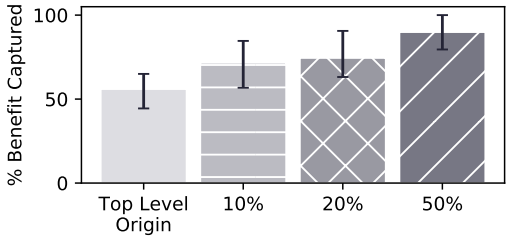


Figure 13: **Percentage of potential benefits achieved when X% of origins in each page run Alohamora. Results are for the {12 Mbps, 100 ms, 2×} setting. Bars show medians, with error bars spanning 25-75 percentiles.**

the cascading effects that those changes have on underlying network resources (e.g., TCP semantics). While similar to Alohamora's simulator with respect to approximate TCP modeling, the core limitation of RT-H2 (with respect to Alohamora's training) is that RT-H2 can only predict the performance of HTTP/2 page loads within the exact same settings as seen in the HTTP/1.1 profiles. In other words, RT-H2 cannot use its profiles to predict HTTP/2 performance outside of the profiling environments. This is problematic for Alohamora's setting, as this implies that a profile would have to be collected for every environment considering during training—this would forfeit most of the simulation benefits. Beyond this, unlike Alohamora, RT-H2 does not consider different performance metrics, compute resources, preload, or variable push policies.

A.2 Other Results

Comparison of existing push/preload strategies: In determining a competitive performance baseline to compare Alohamora with, we considered two standard push/preload (static) heuristics: *push/preload all* and *push/preload all JavaScript*. As noted in §6.1, these strategies solely differ in the set of objects that they consider for push/preload. Using the same experimental setup from Figure 7, we compared these two strategies in terms of speedups that they provide over a default page load. Figure 12 presents representative results. As shown, *push/preload all* provides roughly the same (0-1% more) speedups as *push/preload all JavaScript* at the median, and 5-17% larger speedups at the 95th percentile.

Incremental deployment: Since origins make independent push/preload decisions with Alohamora, we ran experiments to understand how Alohamora's benefits vary with different adoption rates. For each page in our corpus, we ordered the domains in the page according to the fraction of objects that they contribute. We then ran experiments where only the top X% of origins used Alohamora; origins not running Alohamora did not push/preload any objects. We also specifically considered the case where only the top-level origin deployed Alohamora. As expected, Figure 13 reveals that benefits increase as more domains adopt Alohamora. However, simply having the top-level origin can achieve 56% of the potential (i.e., with 100% adoption) median benefits.

Cross-page clustering: To this point, the presented results considered Alohamora models that were trained for a single page (across environments). In order to evaluate Alohamora's ability to train generalizable models across a site's pages, we consider the 100 sites presented in §4 (Figure 4). For each site, we trained a single Alohamora model using only a single (randomly selected) page from each cluster, and evaluated across all of the site's pages. Alohamora's cross-page models are able to achieve within 85-90% of the improvements achieved when training individually on each tested page; this slight degradation comes with the significant benefit of improved training efficiency.

Robustness to input errors: To generate push/preload policies, Alohamora's models ingest a variety of observations that collectively characterize the execution environment. While cache contents require zero approximation to collect, network and CPU measurements can be noisy and hard to report accurately. We evaluated Alohamora's ability to deliver speedups in the face of noisy inputs characterizing network and CPU speeds by considering the following errors: average bandwidth, latency, and loss errors of {1, 2, 3} Mbps, {10, 20, 30} ms, and {0.5, 1, 2}%, and CPU slowdown errors of {1, 2, 4}×. We find that Alohamora's generated policies are largely robust to such errors. For instance, median PLT improvements dropped by only 3.4%, 4.6%, and 3.9% in the {24 Mbps, 20 ms, 2× CPU slowdown} environment with errors of 2 Mbps, 20 ms, and 1%, respectively. Similarly, a CPU slowdown error of 1× resulted in only a 2.6% reduction in PLT improvements.

Additional pages: In addition to the 500-page corpus that we used for our primary experiments (§6.1), we also eval-

	Cluster 1	Cluster 2	Cluster 3
Cluster 1	88% (91%)	50% (55%)	52% (59%)
Cluster 2	57% (52%)	94% (89%)	59% (55%)
Cluster 3	61% (56%)	49% (57%)	100% (93%)

Table 4: **Evaluating Alohamora's cross-page generalization approach. Results are for the {24 Mbps, 20 ms, 2× CPU slowdown} setting, and 30 pages per site. Here we show results for two representative pages that yielded 3 clusters each: NPR (clusters with 19, 8, and 3 pages) and CNN (1, 17, and 12 pages). For each cluster, we picked a random page and found its best policy (via brute force search). We then applied that same policy to the other pages in the same cluster, and to pages in different clusters. Results list the % of possible push/preload benefits for the median page in each cluster, and are presented as CNN (NPR). Takeaway: policies generalize well within clusters (blue regions), but not across clusters (white regions).**

uated Alohamora on two additional sets of sites: 1) 100 interior pages that were collected using a monkey crawler [1] that clicked links on each landing page in our primary corpus, and 2) 100 less-popular landing pages that were randomly selected from the Alexa top 10,000 list (excluding the top 500). Using the same experimental setup described in §6.1, we find that the previously reported trends persist, and in fact, Alohamora's benefits increase. For example, in the {24 Mbps, 20 ms, 1× CPU slowdown, 0%} setting, median PLT improvements with Alohamora were 26% and 27% for the interior and less-popular corpora, respectively; for comparison, the push/preload all strategy achieved benefits of only 3% and 5%. These results are consistent with the observations in prior work [40] that interior and less popular pages are typically more complex, and involve longer serial dependency chains that can be optimized.

Energy usage: To evaluate the impact that Alohamora has on mobile device energy usage, we reused our real world experimental setup (§6.4) and connected the Nexus 6 smartphone to a Monsoon power monitor [35]. Overall, we observed that Alohamora reduces median per-page energy consumption by 23% and 16% compared to a default page load, on the LTE and WiFi networks, respectively. The savings are higher in the LTE setting primarily due to the fact that LTE radios consume more energy than WiFi hardware when active [57]—the higher network latencies on LTE networks lead to more significant load time reductions, which in turn produce larger energy savings.

Clustering: Table 4 presents evaluation results for Alohamora's clustering strategy using two representative sites; these results are a subset of those in Figure 4.

A.3 Additional Related Work

Mobile-optimized pages: Certain sites cater to mobile settings by serving pages that involve less client-side computation, fewer bytes, and fewer network fetches. For example, Google AMP [21, 25] is a recent mobile web standard that requires developers to rewrite pages using restricted forms of HTML, JavaScript, and CSS. Unlike AMP, Alohamora accelerates legacy pages without developer effort. Further, Alohamora's adaptive push/preload policies can improve the performance of AMP pages because all page resources still must traverse a client's slow mobile access link.

Prefetching: Prefetching systems predict user browsing behavior and optimistically download objects prior to user page loads [28, 49, 62]. Unfortunately, such systems have witnessed minimal adoption due to challenges in predicting what pages a user will load and when; inaccurate page and timing predictions can waste device resources or result in stale page content [52]. In contrast, Alohamora generates push/preload policies only after a user navigates to a page, and considers the environmental conditions and page properties collected in situ.