The digitization of economy has had a revolutionary impact on society, and, today, people use digital web services to conduct essential daily activities, such as business, work, education or communication. Consequently, the performance and availability of a web service has a direct impact on society, ranging from impacting business revenue to affecting people’s quality of life. To ensure performant delivery of service workloads (e.g., website code, media objects) and robust availability, service providers typically use Content Distribution Networks (CDNs) that operate geographically distributed server deployments to connect vast areas of the world. At a high-level, a CDN’s web stack comprises several layers, ranging from service code (e.g., HTTP web), networking layers (e.g., transport and application protocols), to the client-side applications (e.g., website JavaScript processed by a generic browser). While maximizing the performance and availability is a key goal, CDNs also aim to minimize the complexity of their infrastructures and traditionally use generalizable solutions, such as same network stack or website design for different services or users, to reduce the CDN operator and web developer effort cost of designing and optimizing the various layers.

Today, Hyperscalers, such as planet-scale CDN infrastructures from Meta, Google and Microsoft, provide diverse web services to billions of users across the world and, with the growth in the scale, a new set of challenges have emerged towards maximizing the performance and availability goals. Hyperscalers are expected to host tens of services with diverse natures (e.g., TCP, QUIC, publish/subscribe), and serve a heterogeneous user-base with diverse last-mile connectivity and device types. Further, the service code experiences a high churn rate and is updated up to tens of times a day. These dimensions directly impact the degree of user satisfaction, commonly called Quality-of-Experience (QoE), as web performance is sensitive to network and devices, while minimizing downtime during service restarts is sensitive to the nature of their protocols. While this heterogeneity is the product of end-user’s choice of connectivity/device and developer’s choice of service/website design; from the CDN’s perspective, the diverse user population and the different implementation of the services end up sharing the same web server stack, network and device resources. Yet, the CDN is required to be consistently performant and available, while maintaining the simplicity of its infrastructure. Effectively, with the limited set of choices at the server-side, i.e., a server stack that can make different choices for the different web stack layers, the CDN is required to optimize QoE for the vast range of heterogeneity that exists outside its purview.

Inherently, a tussle exists between the two contrasting goals of maximizing “QoE” and managing “least-cost”, defined as the smallest number of choices a CDN can make to improve QoE without introducing additional overheads. For instance, while the least-cost approach might be to make no choice and simply used a generalized stack, it may not address the domain-specific needs of the diverse user-base and result in suboptimal QoE. On the other extreme, a CDN may strive to make the optimal domain-specific choices for every individual connection, ranging from tuning the networking stack, adding domain specific enhancements to manage the consistency of diverse protocols during service updates, to re-designing the website to optimize
their resource usage. However, a lack of algorithmic, protocol and system support in the traditional CDN design hinders the practicality of introducing such dynamic flexibility in a low-cost manner: (i) lack of algorithmic support to address the network volatility and the high last-mile/device dimensionality of the Internet makes it challenging to automatically tune protocols in a principled and low-overhead manner, (ii) limitations with kernel and protocols to address the domain-specific needs of non-TCP protocols make it challenging to prevent disruptions for the diverse services, and (iii) limitations with the existing analytics techniques to understand the client-side resource dynamics make it challenging for the developers to identify the key targets to improve in their website design. In light of these challenges, the space of making the right design choice for the different layers is vast, contributing to the high-cost of improving the QoE for the diverse masses.

This dissertation explores the CDN design space to improve the QoE vs least-cost trade-off and proposes “CDPlane: a collection of data-path and control-path components for a flexible CDN”. While a CDN may not have direct control over the source of heterogeneity, the eventual impact of heterogeneity manifests itself in the form of different types of states across the layers, e.g., transport network properties of a connection, application-level state for diverse services, client-side resource allocations for the page load. CDPlane proposes algorithmic, systems, and analytics design components that extract and leverage the state to (i) auto-tune network stack for diverse users by selecting the most suitable protocol configurations based on a connection’s transport and application state, (ii) propose system and protocol abstractions to preserve the connection state during the application update restarts, and (iii) propose a measurement technique to gain memory state visibility into the JavaScript’s interactions with browser layers to identify the key culprits behind a website design’s memory overhead. While the domain-specific design of the algorithm to combat network dynamics eliminates the cost of manually curating networking stack configurations, the proposed system and protocol abstractions for disruption-free updates only extend the traditional designs and do not require a specialized redesign of services. Further, the analytics technique directly identifies the source of memory bloat and uncovers potential avenues for optimizations, reducing the developer effort required to inspect the large space of JavaScript functions, libraries, and their interactions with the browser components. Taken together, these building blocks extend the traditional design to improve QoE for the diverse services and user-base, and optimize the tussle in a low-cost manner.
© Copyright 2023 by Usama Naseer
This dissertation by Usama Naseer is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date                      Theophilus A. Benson, Director

Recommended to the Graduate Council

Date                      Shriram Krishnamurthi, Reader

Date                      Marwan Fayed, Reader

Date                      Aruna Balasubramanian, Reader

Date                      George Konidaris, Reader

Approved by the Graduate Council

Date                      Thomas A. Lewis
                          Dean of the Graduate School
Acknowledgements

Ph.D. is a long and tiring journey, and I was fortunate to have so many incredible people around me to provide support and guidance. First and foremost, I would like to express gratitude to my advisor, Theo Benson, for his continued support. Working with him has tremendously helped me in understanding how to systematically approach, frame and solve a research problem.

I am grateful to my dissertation committee, Shriram Krishnamurthi, Aruna Balasubramanian, Marwan Fayed and George Konidaris, for their insightful feedback that has tremendously helped me in polishing this dissertation. I am also deeply indebted to my industry collaborators, Luca Niccolini, Ranjeeth Dasineni and Udip Pant to name a few, for their incredible support. I would especially like to thank Marcel Flores for his technical and emotional support and guidance. Working with the industry collaborators really helped me in understanding the scale of today's CDNs and the challenges that entails. I am also grateful to the masters and undergraduate students I had the pleasure of working with over the years.

I owe a lot to the friends I made during my time at Duke and Brown. Their emotional support and company was not just essential for keeping my sanity intact, but our frequently thought-provoking discussions led to the exploration of new ideas and broadening my horizons. In this regard, I would especially like to thank Ghulam Murtaza, Numair Khan, Abdul Manan, Amir Ilkhechi, Suhail Imtiaz, Saim Salman and Ayush Bhardwaj to name a few. I would also like to thank my lab-mates over the years for their continued help.

Finally, and most importantly, I would like to express my deepest gratitude to my parents, to whom this dissertation is dedicated to, my wife and my siblings. Words cannot describe the role their support has played over the years. Their belief, love and emotional support has kept my spirits high during this challenging process and I owe everything to them.
Contents

List of Tables xi

List of Figures x

1 Introduction 1

1.1 Challenges with Improving Performance and Availability . . . . . . . . . . . . . . . . . . . 3
1.2 CDPlane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.2.1 System and Protocol Abstractions for Disruption-free Restarts (Zero Downtime Re-
lease) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.2.2 Algorithmic and System Design for Protocol Tuning (Configanator) . . . . . . . . . 9
1.2.3 JavaScript Memory Overhead Diagnosis (JS Capsules) . . . . . . . . . . . . . . . . . . 10
1.2.4 Roadmap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2 Background 12

2.1 Content Distribution Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.1.1 CDN Infrastructure Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.1.2 Request Workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Continuous Release . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.3 End-user Heterogeneity and Performance Dynamics . . . . . . . . . . . . . . . . . . . . . . . 18

3 Motivation 22

3.1 Ensuring Zero-Downtime in the Era of Continuous Release . . . . . . . . . . . . . . . . . . 22
3.2 Ensuring High Web Performance for the Heterogeneous User-base . . . . . . . . . . . . . 25
3.2.1 Protocol Configuration Tuning at Edge . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.2.2 JavaScript Memory Overheads at the Client-side . . . . . . . . . . . . . . . . . . . . 28

4 Zero Downtime Release 31

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2 Background and Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.2.1 Traffic Infrastructure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.2.2 Life of a Request . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Request sub-workflows.</td>
<td>16</td>
</tr>
<tr>
<td>5.1</td>
<td>Web stack configuration parameters.</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>Heterogeneity in configs. across 5 regions</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Impact of NC size on performance.</td>
<td>70</td>
</tr>
<tr>
<td>5.4</td>
<td>Simulator knobs</td>
<td>77</td>
</tr>
<tr>
<td>6.1</td>
<td>Top-10 events and their metadata.</td>
<td>94</td>
</tr>
<tr>
<td>6.2</td>
<td>Memory allocators used by the different browser actions performed during a page load.</td>
<td>95</td>
</tr>
<tr>
<td>6.3</td>
<td>Clusters’ events composition.</td>
<td>113</td>
</tr>
<tr>
<td>6.4</td>
<td>Optimizations.</td>
<td>118</td>
</tr>
<tr>
<td>6.5</td>
<td>Correlation for key event metadata.</td>
<td>126</td>
</tr>
<tr>
<td>6.6</td>
<td>Correlation for key event metadata (continued).</td>
<td>127</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Web stack layers. .................................................. 2
1.2 QoE vs Least-cost curve. ....................................... 4
2.1 CDN infrastructure overview. ............................... 14
2.2 Proxy update restart workflow. .............................. 17
2.3 Web page load. .................................................. 19
3.1 Number of UDP packets mis-routed. .......................... 23
3.2 Number of MQTT connections disrupted. ................... 23
3.3 Percentage of HTTP POSTs failed. ............................ 23
3.4 Impact of configuration tuning on page load time. .......... 27
4.1 End-to-end Infrastructure .................................... 33
4.2 # of releases .................................................. 33
4.3 Root of L7LB restarts .......................................... 33
4.4 Root of app. tier restarts ...................................... 33
4.5 UDP mis-routing .............................................. 33
4.6 Cluster capacity .............................................. 36
4.7 CPU spike ................................................... 36
4.8 Impact of release ............................................. 36
4.9 Design choices for PerfectReload ............................ 37
4.10 Socket Takeover .............................................. 37
4.11 Downstream Connection Reuse .............................. 41
4.12 Partial Post Replay ........................................... 41
4.13 Cluster capacity at {5,20}% batch restarts. ............... 46
4.14 Impact of Downstream Connection Reuse .................. 47
4.15 Packet mis-routing ........................................... 48
4.16 POST disruption ............................................. 48
4.17 Proxy errors comparison ..................................... 48
4.18 Timeline of Proxygen restart ............................... 51
4.19 Disruptions during Proxygen restart ........................ 52
5.37 PLT variability ................................................................. 86
5.38 % sessions with PLT degradation. .............................. 87
5.39 Number of flows through access link ............................... 88

6.1 Average page load and JS memory timeline from 2015 to 2021. .......... 97
6.2 Increase in JS memory for the websites that are consistently present in snapshots. 97
6.3 Duration between consecutive browser events during a page load. .......... 99
6.4 Categorized breakdown of JS triggered events. ..................... 99
6.5 Memory coverage of existing tools. ..................................... 100
6.6 Allocator-wise breakdown of JS memory. ......................... 100
6.7 Coverage vs attribution of existing tools. ......................... 100
6.8 Browser mechanics for executing bar() JS ............................ 101
6.9 JS-Capsules illustration .................................................. 101
6.10 JS-Capsules are created by identifying JavaScript events and appropriate cascading events, then using their timestamps to capture the associated memory allocations. ................. 101
6.11 Elbow method for K selection. ........................................ 104
6.12 JS-Capsule vs ground-truth JS memory. ......................... 108
6.13 Number of capsules for different devices. ......................... 108
6.14 Number of functions per JS-Capsule. ............................. 108
6.15 Impact of garbage collection on measurement noise. ............... 108
6.16 RF prediction error. .................................................... 108
6.17 Correlation for key events. ............................................ 108
6.18 Memory allocation clusters. ........................................... 111
6.19 Cluster presence across websites. ................................... 111
6.20 Capsule’s individual versus cumulative memory size. ............... 111
6.21 Source of V memory. .................................................... 112
6.22 JavaScript workflow and scope of optimizations. ................. 117
6.23 Impact of optimizations on JS memory. ............................. 119
6.24 Source of optimization’s savings. .................................... 121
6.25 Extent of unused JS code. ............................................. 122
6.26 # of functions that are eagerly parsed. ............................. 122
6.27 Produced cache size. .................................................... 122
6.28 R memory impact of DOM complexity. ............................. 127
6.29 Comparison of 50 random and Alexa top 1K websites (median cluster memory contribution) 129
6.30 Comparison of 50 random and Alexa top 1K websites (tail p95 cluster memory contribution) 129
6.31 # of nodes in DOM trees. .............................................. 131
6.32 Tree depth. ............................................................... 131
6.33 # of nodes as direct children of HTML head/body. ............... 131
6.34 # of nodes parented by the top 50 nodes. .......................... 131
6.35 HTML tag type distribution. ......................................... 131
Chapter 1

Introduction

The digitization of economy has had a tremendous impact on society and has revolutionized the way people conduct their businesses, communicate with each other and perform a wide range of essential daily activities. Studies show that digital services have improved people’s quality of life [423], enabled them to grow their businesses [473, 164, 370], and have ushered in an era of “Digital Revolution” that has reportedly elevated millions out of poverty in the developing regions [407]. Consequently, today, digital services have become a part and parcel of modern life, as depicted by the role that digitization of classrooms, workplaces, and entertainment played in enabling people to reclaim some semblance of normalcy and the continuity of business operations, during the recent COVID-19 pandemic.

Due to their pivotal role in modern life, it is important for the digital services to maintain and improve the degree of user satisfaction, i.e., Quality-of-Experience or QoE. Degradation in QoE, either due to poor performance or service downtime, has multi-faceted implications on society, ranging from disruptions in users’ lives to revenue losses for the businesses. A study from the U.K. reported that 89% of the surveyed remote workers lost 6.3% of their daily productivity due to poor web performance [392], while 50% of participants in an Ericsson study experienced 19% elevation in stress-levels due to a 6s delay in video loading [423]. Further, user engagement and revenue are directly tied to QoE, motivating businesses to improve their service QoE, e.g., BBC reported a loss of 10% of users for every one second inflation in page load time (PLT) [109], while others reported significant gains as a result of improving service QoE: revenue improved by $530,000 for every 100ms decrease in PLT for Mobify [473], 15% increase in sign-ups by 40% improvement in perceived wait times for Pinterest [164] and 7% increased conversions for a 850ms PLT improvement for COOK [370]. The financial outcomes of poor QoE can be so severe that it can put companies out of business [13]: while an average business loses ~$300,000 per hour due to service downtime [13, 299], the consequences are dire for larger enterprises, with tens to hundreds of millions lost for every hour of downtime [48, 356].

To improve performance and minimize downtime, service providers typically employ the robust and efficient networking infrastructure provided by Content Distribution Network (CDNs). CDNs operate geographically distributed server deployments that connect vast areas of the world, cache web content across regions, and enable fast and reliable delivery of web content to the end-users [318, 186, 359, 358, 463].
Today, hundreds of commercial CDNs are available [318] that differ in their networking and system designs [194, 359, 358, 295, 314], and the different designs were largely adopted in response to the evolving scale, business and performance requirements. Among these designs, Hyperscaler CDNs — the key focus of this thesis — hold a unique position in the design space. Hyperscalers, such as the content distribution platforms from Meta, Google and Microsoft, were born out of necessity to tackle the growing scale of the diverse services, and differ from other CDN designs in their scale and degree-of-control over the end-to-end infrastructure. In contrast with the other CDN designs, for Hyperscalers, a single organization owns the infrastructure and workloads, thereby granting a complete operational control. On one hand, the ownership ensures sovereignty and security [241], while on the other, the control over the end-to-end infrastructure allows the design of novel hardware, software and networking solutions to serve one’s domain-specific scale and performance requirements [33, 265, 143, 142, 91, 84].

At a high-level, the end-to-end CDN infrastructure comprises several layers shown in Figure 1.1 (referred to as web stack layers). At the server-side, the stack comprises service code (e.g., www, publish/subscribe) that runs on server applications such as apache, nginix, node, etc. The server apps. communicate\(^1\) with the clients over wide-area network (WAN) through a networking stack that comprises several layers, such as L7 (application) and L4 (transport). For the networking layers, the CDN has an option to choose from a plethora of protocols, such as TCP or UDP transport, different flavors of TCP congestion control, different HTTP versions, and cross-layer protocols like QUIC. On the other hand at the client-side, the web stack comprises web applications, primarily browsers, that interact with the services, run client-side service code (e.g., JavaScript) and load the services (e.g., webpages). The browser comprises several layers or components [107, 283], such as V8 to run JavaScript [502], compositor to render webpage [285], GPU for graphical tasks [494], and these components interact the the OS/hardware layers for network, computational and memory resources. While the Hyperscalers are able to exercise control over all the server-side layers, their control may be limited on the client-side layers\(^2\), e.g., control only limited to the service code (e.g., JavaScript) for services made to

---

\(^1\)CDNs typically divide their application stack across two tiers. Server apps. are hosted in a data-center/origin tier, while proxy servers at edge tier handle client connections and serve as a relay between server apps. and clients.

\(^2\)A higher level of client-side control is available for special-purpose applications, e.g., Facebook Messenger, Gmail, Office 365, where Hyperscalers design the application from the ground up.

---

**Figure 1.1:** Web stack layers.
work with generic browsers.

The web stack layers directly dictate the QoE, for instance, the choice of congestion control impacts the delivery performance [445, 433], website JavaScript design and the client-side device resources impact the processing speed [373, 123], etc. Consequently, the eventual QoE observed by the user population depends on the proper functioning of the server-side and client-side layers, e.g., optimal page load QoE requires efficient data delivery by the networking stack and an efficient design of client-side website code to make the best use of device resources. Moreover, the responsibility to optimize the different layers may lie across different entities: 3rd-party content-providers (e.g., HBO, HULU or ESPN) or specialized web teams for Hyperscaler (e.g., www team at Meta) design the services and, consequently, the task of optimizing the service code (e.g., website) is generally delegated to the developers who have the domain-specific knowledge of their design. However, Hyperscalers can leverage their control to employ various analytics infrastructures, such as Odin [91], NEL [84], Conviva [117] or client-side monitoring [56, 369, 1, 134] to provide developers with the tracing information to help them improve their web design.

1.1 Challenges with Improving Performance and Availability

Hyperscalers strive to maximize their performance and availability, while minimizing the cost and complexity of their infrastructure [314, 91, 463]. However, owing to the growth in the scale and the complexity of services, today, they face major challenges in meeting these goals. Hyperscalers host tens of services with diverse natures (e.g., TCP, QUIC, publish/subscribe), and serve a heterogeneous user-base with diverse last-mile connectivity and medium-of-access to the web (i.e., device types), e.g., the developing region users are significantly divergent as compared to their developed regions’ counterparts (2G and 4G/5G hold 37% and 12% share in Sub-Saharan Africa, while ≤1% and 90% in North America, respectively; low-end devices still hold a strong share in developing regions [20]). Further, a service’s code experiences a high churn rate to minimize the Time-To-Market for feature, security and performance upgrades [266, 389] and, consequently, production services are updated up to tens of times a day [122, 440, 240, 121]. These heterogeneous dimensions directly impact the QoE aspects of the web stack layers, since optimal performance requires careful tuning of the networking stack to cater for the unique characteristics of the diverse networks [520, 165, 27, 383, 305] and a principled design of client-side web applications to improve the efficiency of resource usage [32, 168, 509, 518, 371, 123]. Further, smoothly deploying service updates (i.e., restarting production application servers to update code binaries) without downtime and disruption risks require system and protocol abstractions to address the domain-specific requirements of service protocols [332, 323].

While the user-base and service heterogeneity is the product of end-user’s choice of connectivity/device and developer’s choice of service/website design, from the Hyperscaler CDN’s perspective, the diverse user population and the different implementation of the services end up sharing the same web server stack, network and device resources. Yet, the CDN is required to be consistently performant and available, while maintaining the cost-effectiveness and the simplicity of its infrastructure. Effectively, with the limited set of choices at the server-side, i.e., a server stack that can make different choices for the different web stack layers, the CDN is
required to optimize QoE for the vast range of heterogeneity that exist outside its purview.

Inherently, a tussle exists between the two contrasting goals and, in Figure 1.2, we present an abstract representation of the tussle between “least-cost” and “QoE”. We define “least-cost” as “the smallest number of choices a CDN can make to improve QoE without introducing additional overheads”, while the “QoE” dimension captures the degree of satisfaction with the CDN services for the user-base. Naturally, the different web stack layers shown in Figure 1.1 impact the QoE in different ways, e.g., while service disruptions and downtime may render the service unavailable to the users, network congestion and client-side resource overheads may lead to inflation of page load times.

In practice, an inverse relationship exists between the two goals, for instance, while the least-cost approach might be to make no choice and simply used a generalized stack, it may not address the domain-specific needs of the diverse user-base and result in sub-optimal QoE. On the other extreme, a CDN may strive to make the optimal domain-specific choices for every individual connection, ranging from tuning the services and the networking stack, to asking the web developers to re-design their websites to optimize their resource usage. While this may result in higher QoE, such a choice is either impractical or high-cost in practice due to limitations with system, protocol, algorithmic or analytics support, as discussed below:

**Challenges and design choices for continuous release:** At the Hyperscaler scale, the release mechanisms are required to be swift (i.e., tens of updates per day) while simultaneously being disruption-free (i.e., high QoE) and incurring minimal cost (i.e., highly generalizable and low resource requirements). However, such a design face several challenges. First, Hyperscalers employ a diverse range of protocols and services, thus, requiring domain-specific maneuvers to prevent service-specific disruptions, e.g., TCP vs QUIC transport protocols and HTTP vs MQTT (for publish-subscribe [357]) app. protocols have distinctly different tolerance and state requirements. Second, while many applications are stateless, a non-trivial set of applications are stateful, thus requiring either seamless persistence across restarts or transparent recreation of state at the updated server, e.g., a non-trivial number of connection are long-lived and failing to preserve the state degrades user QoE due to connection terminations during restarts. Third, due to application-specific requirements, a subset of servers are resource-constrained (e.g., cache priming [207, 39] for web servers [6] consumes most available memory), which puts additional constraints on the nature of mechanism that can be used.
Hyperscalers have several mechanisms in their arsenal, ranging from low-cost/high-disruptions to high-cost/minimal-disruptions. Today, most providers adopt either blue/green deployment or rolling updates due to their simplicity: the former relies on over-provisioning by maintaining two identical environments where one runs last-stable version and the other runs newer code, while the latter incrementally updates the applications in small batches. Both mechanisms allow a restarting instance to gracefully conclude existing connections, for a duration called draining period, and then forcefully restart a newer instance of the application with the updated binary. If a connection is not gracefully terminated during the draining period, it is abruptly shut down and the client is required to setup the connection from scratch (e.g., request retries). While being simple, these mechanisms leads to a host of undesirable consequences, from lowering aggregate server capacity for the draining period to terminating connections that outlive the draining period. At the scale of billions of connections, such disruptions are disastrous for the ISP, end-user, and the CDN.

Orthogonally, there are efforts to hot-restart production servers, where a server instance with new code is spawned and takes-over the TCP listening sockets, leveraging kernel features. However, as we further show in Chapter 3.1, such mechanisms do not address the domain-specific needs of non-TCP services and lead to connection terminations in order of millions per restart, e.g., hot-restart works on the assumption that the kernel holds the connection state, that is violated for QUIC services where applications hold the respective state. Further, such mechanisms also lead to disruptions for specialized services, such as publish/subscribe or HTTP uploads where connections are long-lived and require state perseverance across restarts. Though, one may opt to preserve and migrate app-level state to healthy instances, however at a scale of millions of connections, the current literature fails in providing such a mechanism in a low-overhead manner. On another extreme, there are also efforts to design languages with built-in support for headless updates such as Erlang. However, this approach is not supported by most common languages and requires an enormous developer cost to re-engineer the legacy applications in these specialized languages.

**Challenges and design choices for protocol tuning:** Traditional networking stacks do not support protocol tuning on a per-connection basis for the wide range of TCP and HTTP configurations and, consequently, prohibits flexible protocol tuning. Even if such support is available, the key challenge with dynamic protocol tuning lies in designing the control algorithms that can systematically control the network stack flexibility. At the Hyperscaler scale, the design of such algorithms is challenging due to the QoE-cost (i.e., degradation) of testing a sub-optimal configuration and the QoE sensitivity of various connection features across the different layers such as network volatility (e.g., changes in connection latency, bandwidth or packet loss rates), application (e.g., website complexity, number and types of objects) and the high last-mile/device dimensionality of the Internet.

Most attempts for protocol tuning involve manually analyzing the performance of configuration options across different networks, devices, or websites, and selecting the configurations that maximize performance for the median user. However, such hand-tuning approach incurs a high-cost at the operator-end to manually analyze possibly thousands of configuration combinations for the high-dimensional Internet (diverse last-mile connections, devices and websites). Further as network conditions are dynamic and change over time, such static “one-size-fits-all” approaches do not evolve over time and are thus unable to maximize QoE for the diverse population. On the other hand, others have proposed offline
modeling techniques [25] and online statistical algorithms [404, 29, 300, 151, 504] for tuning system configurations. While these approaches offload the selection of configurations to automated techniques (thereby reducing the manual effort), they do not address the domain-specific requirements of edge protocol tuning, i.e., high-QoE cost of wrong configurations, network dynamics and the high dimensionality of the Internet. Consequently, they result in sub-optimal QoE, as we show in Chapter 3.2.1 where we perform a thorough evaluation of the existing techniques to evaluate their cost and QoE dynamics.

Challenges and design choices for JavaScript memory optimizations: Optimizing resource usage of websites — specifically JavaScript memory, the key focus of this dissertation — requires analytics techniques to capture fine-grained allocations (i.e., how much memory is allocated?) and identify their source (i.e., what and why do the JS functions incur memory overhead?). The two properties tackle the challenge of understanding JavaScript memory from two different angles: while the former enables a developer to accurately measure function-wise memory and identify the problematic functions, the latter provides visibility into the nature of JavaScript and their impact on browser mechanics (i.e., browser events), essential for understanding the source of allocations. However, realizing both of these tasks in practice for an arbitrary website comes with a significant amount of challenges. First, a JavaScript function often leads to a wide range of browser events that span across multiple browser layers, from JavaScript engine V8 [502], networking components, rendering engine [215] to the GPU [494] and thus incurs a direct and an indirect memory overhead. Second, identifying the key root-cause of memory overhead, i.e., the specific browser mechanics and their properties involved in parsing, compiling and executing a function, is complicated due to the fine-granularity of events and the wild-wild-west nature of JavaScript. Consequently, a holistic JavaScript memory overhead diagnosis requires a cross-component and fine-grained memory view to attribute the memory overhead to the respective JavaScript functions and their cascading events.

Unfortunately, the existing recommendations solely rely on OS-level [154, 405] or Chrome-based [137, 491, 11] memory analysis tools that either provide incomplete coverage or coarse-grained visibility. For instance, while HeapProfiler [11], recommended in [62, 272], measures memory on a functional-basis, it only focuses on the JavaScript heap. Consequently, its scope is limited to the V8-component, thus providing incomplete coverage as our measurements show that V8 only contributes \(\sim 27\%\) of the JavaScript memory at median. On the other hand, while MemoryInfra [139], recommended in [137, 345], measures memory across all the components, it does not measure JavaScript memory in isolation. Particularly, it measures the complete overhead incurred during the page load and does not tie back the cross-component allocations to fine-granular JavaScript functions, thereby, making it challenging to infer if allocations across the V8 boundary are an artifact of JavaScript indirect memory overhead.

In absence of such techniques, the scope of optimizations is vast, requiring high-cost at the developer-end to understand the resource overhead that may arise due to a variety of reasons, such as JavaScript inefficiencies, resource-heavy libraries, DOM design flaws etc. Consequently, improving the QoE requires high-cost effort at the developer-end to analyze thousands of JavaScript functions and their fine-grained interactions with the browser layers to, first, identify the source of overhead and, second, identify or design an optimization that may be helpful.

In light of these challenges, the space of “making the right choice” is vast, contributing to the high-cost
of improving the QoE for the diverse masses. Consequently, the traditional designs are limited in their ability to adapt according to the needs of diverse operating conditions.

1.2 CDPlane

This dissertation explores the CDN design space to improve the QoE vs least-cost trade-off that arises due to heterogeneity outside of a CDN’s purview. While a CDN may not have direct control over the source of heterogeneity, the eventual impact of heterogeneity manifests itself in form of different types of state across the layers, e.g., transport network properties of a connection, application-level state for diverse services, client-side resource allocations for the page load. While it might be trivial to extract such state from some of the layers, e.g., the edge server can infer network properties such as network latency from the transport layer, it can be quite challenging for other layers and requires system or protocol mechanisms to extract or indicate a change in state, e.g., application-level state for QUIC and publish/subscribe services, client-side cross-layer memory allocations for JavaScript.

Our key insight is: “End-user connections and page load requests span multiple layers in the web stack and key information about the nature of connections, performance sensitivity of protocols and client-side resource usage can be inferred from the cross-layer state. When the CDNs use the cross-layer state properties to manage connection consistency, tune protocols or inform the design decisions for updating website code, they can extend their traditional designs to improve QoE without significant additional cost”. Based on this insight, the dissertation proposes “CDPlane”: a collection of data-path and control-path design optimizations.

CDPlane is composed of algorithmic, systems and analytics design components that extract and leverage the state to (i) auto-tune network stack for diverse users by selecting the most suitable protocol configurations based on a connection’s transport and application state, (ii) propose system and protocol abstractions to preserve connection state during the application update restarts, and (iii) propose a measurement technique to gain memory state visibility into the JavaScript’s interactions with browser layers to identify the key culprits behind a website design’s memory overhead. While the traditional designs operate within the context of individual layers due to the modularity and isolation principles, CDPlane trades-off generality in favor of finding a better cost-QoE trade-off. While the domain-specific design of algorithm to combat network dynamics eliminates the cost of manually curating networking stack configurations, the proposed system and protocol abstractions for disruption-free updates only extend the traditional designs and do not require a specialized redesign of services from the ground up. Further, the analytics technique directly identifies the source of memory bloat and uncovers potential avenues for optimizations, reducing the developer effort required to inspect the large space of JavaScript functions, libraries and their interactions with the browser components.

Next, we describe the details of the three main contributions, discuss the key insights and discuss how the designs achieve a better QoE vs least-cost trade-off.
1.2.1 System and Protocol Abstractions for Disruption-free Restarts (Zero Downtime Release)

Hyperscalers deploy massively complex code-bases on large sprawling infrastructures to deliver rich web services to billions of users. These code-bases are constantly modified to introduce bug fixes, performance optimizations, security patches, new functionality, among a host of other reasons, and tens of thousands of commits are pushed to tens of thousands of machines across the globe, each day [440, 427]. Consequently, production server and proxy applications are restarted to update the code binaries up to tens of times a day and pose a downtime and disruption risk.

In this project, we propose a set of mechanisms to prevent disruptions at large-scale due to update restarts. Our key insight is that the diverse services make distinctly different state requirements from the infrastructure, e.g., TCP maintain connection state within kernel, QUIC within application; state for publish/subscribe connections reside within back-ends, etc, and disruptions can be avoided if domain-specific maneuvers are introduced to maintain state consistency during the restart. Our framework builds on the following two principles: first, as Hyperscalers control the end-to-end infrastructure, we can leverage the hierarchy within the different web stack layers to localize the disruptions within the CDN and prevent them from reaching the end-users (i.e., making restarts transparent to the end-users), and second, while it may be costly to migrate the state (e.g., for QUIC connections) or re-fetch the state through request re-tries (e.g., for long-lived uploads or pub/sub connections over high-RTT WAN), we can add light-weight service-specific system and protocol enhancements to add fail-over data-paths that either maintain cross-layer stat consistency by forwarding connections to application instances with the respective state, or in-expensively rebuild the state at the CDN-side without any user interactions.

Our framework, **Zero Downtime Release**, achieves disruption-free releases by signaling the upstream tier to handle connection migration and by indirectly externalizing user-specific state and redirecting the state to an upstream tier. The upstream tier redirects existing connections and externalizes the state to the new servers (or servers with the new code). Additionally, zero downtime for a restarting L7LB is achieved by handing-over traffic to side-car (instance with the new code) on the same machine. To this end, our framework consists of three mechanisms: a technique for signaling and orchestrating connection hand-off via an upstream tier, a method for detecting and externalizing state, and enhancements to pre-existing hand-off kernel-based mechanisms to enable them to scale to billions of users.

**Zero Downtime Release** framework has been deployed at Meta for several years and has helped to sustain an aggressive release schedule on a daily basis. On the QoE vs least-cost scale, the framework excels the traditional methodologies, providing better QoE without introducing significant additional cost. Our comparison with previously used release methodologies shows that **Zero Downtime Release** leads to (i) improvement in L7LB CPU capacity by 15-20%, (ii) prevention of QUIC disruptions that are in the order of tens of thousands of connections per proxy instance for the traditional methodologies, and (iii) prevent millions of error codes from being propagated to the end-user. On the cost front, the framework only conservatively extends the existing infrastructure by introducing light-weight systems and protocol abstractions for signaling the intent to restart and fail-over data-paths. Our design does not require the high-cost of redesigning new server/proxy applications with specialized languages, redesign new protocols or incur significant additional
resource overheads (~4% higher CPU and memory overhead at median).

### 1.2.2 Algorithmic and System Design for Protocol Tuning (Configanator)

Society’s evolving expectations for better QoE have transformed the networking stack, resulting in the design of “better” congestion control algorithms (CCAs) [149, 95, 543], application protocols (e.g., HTTP version) [520] and a plethora of configuration options to deliver higher performance [152, 363, 27, 187] (Table 5.1). However, uniformly improving web performance is becoming increasingly challenging due to the growing disparity in the network conditions (e.g. bandwidth, RTT) [464, 520, 27, 165] and end-user devices [366, 429, 519, 371], as the optimal choice of configurations is contingent on the network [520, 165, 464, 27, 383, 543, 305], website complexity [85, 88, 375, 521, 519], and end-user devices [20, 429, 371]. Consequently, optimal performance requires careful tuning of protocol and configuration options to cater for the diverse needs of the divergent user connections.

In this project, we eschew the notion of a homogeneous approach to tuning web server configurations and instead argue for a “curated” approach for configuring on a per-connection basis (called Configanator). The design of Configanator’s algorithm is an embodiment of domain-specific insights to tackle network dynamics, high-QoE cost and high-dimensionality, and is designed explicitly to continuously converge to a near-optimal configuration within a minimal number of exploration steps. At a high-level, we leverage the transport-layer connection state (e.g., latency, goodput) and application-layer state (e.g., device type, website, QoE) to cluster similar connections and build fine-grained performance models that preserve the high-dimensionality of the user-base. Leveraging such models, we introduce a contextual multi-armed bandit [513] algorithm that simultaneously operate in two modes depending on the “quality” of the performance model: (i) reduces the likelihood of testing sub-optimal configurations by intelligently selecting samples that speed up model convergence and amortizes exploration cost to the entire clusters, and (ii) at steady-state, transitions into a greedy-mode that stochastically samples points to iteratively improve performance.

To practically implement the configuration tuning in the web stack, Configanator further proposes a design of networking stack where a single server instance can use different configurations for different connections, and the design of an end-to-end, data-driven system for dynamically tuning the cross-layer networking configurations. Taken together the learning algorithm and system design, Configanator enables a CDN to systematically explore heterogeneity in a dynamic and fine-grained manner while improving end-user performance.

On the QoE vs least-cost scale, Configanator outperforms the traditional approaches on both dimensions. To demonstrate the benefits, we used a private dataset from a global CDN and public datasets from CAIDA [89], MAWI [45], Pantheon [543] and FCC [181] to conduct large-scale simulations, as well as, deployed a live prototype. Our results show that Configanator consistently beats the traditional approaches and provides 32-67% (up to 1500ms) improvement in the PLT at tail (p95) across the different traces. On the cost front, the design of Configanator’s algorithm is specifically curated to address the domain-specific needs of Hyperscaler-scale configuration tuning and the automated nature of our algorithm eliminates the manual effort required from the CDN operators. Further, we show that our approach to configuration tuning does no incur significant overhead. Taken together, Configanator significantly improves QoE for the diverse user-base
without putting additional burden on the operators or the infrastructure to pay a high cost for making the right configuration decisions.

1.2.3 JavaScript Memory Overhead Diagnosis (JS Capsules)

Mobile phones have become the primary gateway to the web, with the number of mobile subscriptions surpassing the total human population by 1.3 Billion [524] and contributing over 55.8% of website traffic worldwide in 2022 [110]. However, despite the importance, mobile web is still plagued with performance issues [87, 372], primarily due to the friction between the high resource demands of the complex websites [86, 12] and the modest hardware of mobile phones [518, 168, 12]. Recent studies from academia have tackled the resource limitations in mobile devices from different axes, including network [167, 379, 280], computation [518, 372, 123], energy [94, 126, 489], JavaScript inefficiencies [99, 168, 280, 98, 290] and, to some extent, web memory overhead [123, 126, 416, 269]. Specifically for the web memory overhead, Chrome, V8 and web development teams from multiple organizations have proposed several recommendations and optimizations, such as optimized JavaScript parsing, compilation and execution [413, 394, 112, 486, 162], object compression [503, 411], heap memory leaks [62, 388, 273, 272] and lazy loading/rendering [17, 418, 68, 177, 178]. Further, others have proposed “signals” to detect memory issues, such as device hardware info [515] and real-time heap memory inference [369, 17], in order to provide developers with toolkits to detect and improve the memory performance of their applications.

While these recommendations have been instrumental in shaping the state-of-the-web, today, we lack techniques that can enable a developer to holistically quantify and qualitatively understand the memory overhead of their website code, specifically JavaScript, a cornerstone of today’s dynamic and interactive web ecosystem [398, 99, 168, 280, 98]. In absence of such techniques, a momentous developers effort is required to inspect the large space of JavaScript functions, libraries and their interactions with the browser components. Such techniques are also essential for providing the developers the context to identify the key reasons behind the JavaScript memory bloat, an understanding crucial for systematically selecting the appropriate optimizations to fix website-specific memory problems or, if none exist, revise and engineer new solutions to fit their domain-specific needs. In fact, Chrome teams have written multiple guides to identify and fix memory problems [62, 272, 137, 345]. However, in our experience, these recommendations either lack the coverage or fine-granularity required to tackle the challenges with JavaScript memory accounting.

In this project, we propose a novel measurement framework for efficiently and accurately capturing fine grained measurements of JavaScript functions’ memory overhead. The key insight behind our design is that most browsers execute JS in a single-threaded fashion [518, 330, 376] per frame and use a single main thread for scheduling JS events [330] to guarantee safe concurrency for the shared DOM data structure. Leveraging this property, we decompose a website’s JS into isolated blocks called JS-Capsules, formally defined as a set of one or more JS functions, represented by a unique, non-overlapping time interval (i.e., start and end timestamps). Leveraging the timestamps and the serial-access model, we capture a JS-Capsule’s cross-component cascading events by decomposing the page load process along a browser’s thread boundaries, and use low-level Chromium APIs [134, 135] to measure fine-grained, cross-component memory allocations. In essence, JS-Capsules provide us with a framework for fine-grained visibility into the JavaScript functions
and the resulting cascading events, as well as, a complete coverage over the various sources of memory allocations. Leveraging the fine-grained information from JS-Capsules, our framework uses statistical and machine learning techniques to identify and understand the JavaScript memory allocation patterns across a wide range of websites.

Leveraging the framework, we present the first-ever characterization of JavaScript memory for the Alexa top 1K websites and show sensitivity of JavaScript memory overhead to a diverse range of features such as V8’s parsing behavior, website’s DOM properties and dynamic interactions between JavaScript and DOM, among others. A key significance of our measurements results is that it unveils the potential optimization targets, if one aims to optimize JavaScript memory beyond V8’s context. JS-Capsules directly help the developers in reducing their search space: it identifies the functions with highest memory overhead and provide developers the contextual information (i.e., browser events) to understand the source of the overhead. Further, as we show in our analysis of websites and popular optimizations, the information provided by JS-Capsules is instrumental for the developer to select the right set of optimizations for their website. Leveraging this framework, we expect the developers to gain deeper insights into their website JavaScript and design domain-specific optimizations to reduce the memory footprint of their websites and improve the QoE for their user-base.

1.2.4 Roadmap

This document is structured as follows: Chapter 2 presents background on different CDNs designs, the growing heterogeneity in code, network connectivity and end-users devices. Chapter 3 discusses the major availability and performance challenges for Hyperscalers and makes a case for adding cross-layer flexibility into the Hyperscaler infrastructure. Chapter 4 discusses our work for minimizing downtime and disruptions during updates releases for diverse services. Chapter 5 discuss system and algorithmic design to automatically tune the configurations in a low-overhead, principled manner, while Chapter 6 discusses a novel measurement technique for memory analysis of client-side JavaScript. Chapter 7 discusses the applicability of the contributions to different types of CDNs, while Chapter 8 summarizes the recent works in CDN optimizations, continuous release updates, auto-tuning systems and web optimizations. Finally, Chapter 9 concludes the dissertation and discusses some future work directions.
Chapter 2

Background

2.1 Content Distribution Networks

Content Distribution Networks or CDNs are geographically distributed clusters of servers that cache popular web content (e.g., HTML pages, JavaScript files, images, video). The server deployments are strategically placed close to large clusters of population, and improve end-user QoE by minimizing latency and improving throughput. Due to their performance and availability benefits, it is often highly recommended for websites to leverage CDNs to improve user QoE, grow their user-base, and cut development costs [77]. In fact, measurements indicate that more than 41% of the top 100K websites either operate their own or use a 3rd-party CDN [318].

CDNs today have adopted a variety of designs and deployment models, and are broadly categorized into three classes: Commercial, Meta, and Hyperscaler CDNs. The key difference between the classes lies in the ownership and control of the end-to-end infrastructure (e.g., network, servers, client-side) and web workloads (e.g., websites, video).

For commercial CDNs, such as Akamai, Cloudflare, and EdgeCast, the ownership and control is limited to the network (e.g., Autonomous System managed by Akamai) and the edge server deployments. Web workloads are produced and licensed by 3rd-party content providers (CPs), such as Hulu, HBO, and ESPN, and the CPs pay a fee for the delivery of their content to the end-users. To optimize delivery and reduce costs, commercial CDNs have adopted a variety of designs, including micro-datacenters operated by commercial CDNs in ISP’s infrastructure [194, 294], purely ISP-operated CDNs [194], hybrid peer-to-peer delivery [18, 534], and CDN federations [295, 386]. However, these designs often require either additional control or collaboration between different entities, e.g., P2P designs require specialized client-side software such as Akamai’s NetSession [18], ISP-related designs either require licensed software from CDNs or leased rack-space from ISPs [194], and federated CDNs require coordination between multiple CDNs [194, 295, 386].

Meta-CDNs, such as Conviva, Cedexis, NicePeopleAtWork, allow CPs to simultaneously use multiple commercial CDNs and provide “broker” services to dynamically select CDNs, based on performance and cost objectives [359, 358, 194]. While Meta-CDNs do not own network, edge servers or workloads, they
build frameworks for measuring QoE from client-side applications (e.g., video players), and leverage data across multiple sessions to build predictive models for data-driven CDN selection. CPs may also opt to implement and run the broker services themselves [314].

Hyperscaler CDNs, such as content distribution platforms from Meta, Google, Microsoft, Netflix, occupy a unique position in the CDN design space. For Hyperscalers, a single organization owns the infrastructure and workloads, thereby granting a complete operational control. While their design can incorporate the optimizations discussed for commercial and meta CDNs, the end-to-end control allows a unique opportunity for a broader range of optimizations that may not be otherwise practical, such as application-specific CDNs (e.g., used for Youtube and Netflix [194]), tighter coupling of infrastructure and workloads (e.g., Google's AMP [33, 34], Facebook’s FreeBasics [142]), direct control over workloads (e.g., light-weight webpages [169, 143, 142]), and direct control over web applications (e.g., better performance monitoring [91, 84]). The control also allows Hyperscalers to deploy diverse a diverse range of services to serve their domain-specific needs.

2.1.1 CDN Infrastructure Overview

Next, we provide an overview of Hyperscaler CDN infrastructure — the key focus of this thesis. Hyperscalers provide a diverse set of services, ranging from web (e.g., hosting, uploads), video (e.g., on-demand, live streaming), front-end services (e.g., request routing, DDoS protection), domain-specific services (e.g., publish/subscribe), to edge compute. Supporting these services while maintaining availability and performance requires a robust and distributed infrastructure.

**Multi-tiered Infrastructure**: The infrastructure of a typical Hyperscaler is divided into three tiers shown in Figure 2.1: (i) *origin-tier* houses the back-end application servers, (ii) *edge-tier* fronts the origin and operates HTTP proxy servers that cache content and relay the requests/responses to and from the back-end servers, and (iii) *client-tier* is responsible for requesting and processing the web workloads.

The origin-tier comprises data-centers (DCs) and supports thousands of services [402, 196]. These facilities are built on *warehouse-scale computing* principles, and serve as a massive reservoir of storage and computation [60]. Hyperscalers typically operate tens of origin DCs, with tens of thousands to millions of servers in each deployment.

The edge-tier comprises geographically distributed clusters of servers (called *Points-of-Presence* or *PoPs*) at the edge of the CDN’s network. These deployments are strategically placed at peering points close to large clusters of population, to cache and serve the web content at close proximity to the end-users. The edge-tier also provides an additional layer of caching, called *origin shield* [453, 410, 160, 159], that enables one PoP to fetch content from another PoP and minimizes the overhead of contacting distant origin DCs. Besides caching and serving requests, edge-tier also operates a number of diverse services, including security (e.g., DDoS protection, firewalls), business logic (e.g., authentication, paywalls), and media processing (e.g., video optimizations) [41]. Hyperscalers operate hundreds of edge deployments, with tens to hundreds of servers in each deployment [194, 101]. Hyperscalers often also deploy small cluster of machines in collocation facilities [41] and 3rd-party ISPs [194, 294] to extend their edge closer to the end-users, e.g., Facebook Network Appliance (FNAs) [175, 69], Akamai’s on-net deployments [21], Netflix Open Connect [374, 184].
By partnering with ISPs, these deployments enable efficient and localized delivery of content, and reduce load from the ISP’s backbone network.

The client-tier comprises web workloads and their processing at the client-side applications. Hyperscalers typically retain complete control over the design of their web workloads (e.g., webpages, video). Although the design of most workloads is compatible with diverse client-side applications (e.g., webpage can be loaded at 3rd-party browsers), Hyperscalers often also build their specialized client-tiers where they have complete control and visibility into the application code, e.g., mobile applications from Facebook, Netflix or Google, in-app browsers, etc.

Networking Infrastructure: Figure 2.1 presents a high-level overview of the different networking components in a CDN. The network infrastructure of the origin and edge tiers comprises routers, network load-balancers for the transport and the application layers, and application servers. The different components serve their domain-specific purposes, ranging from peering, managing traffic at the different layers, caching the content, to serving the requests. Next, we provide a brief overview of the different components and their interactions in the end-to-end infrastructure.

- Routers at the edge serve as the gateway for traffic and peer with ISPs to connect the infrastructure to the external world. Layer-4 LBs (L4LB) [9, 161] serve as a bridge between the network routers and the Layer-7 components (e.g., proxy servers). Routers typically use ECMP [232] to evenly distribute packets across the L4LB layer, which in turn uses load-balancing algorithms (e.g., consistent hashing [9, 161]) to distribute the packets across a fleet of Layer-7 LBs (L7LB) [468, 545]. The L4 and L7 LBs maintain an updated view of the in-service upstream components (L7LBs and application servers respectively) by constantly exchanging health-checks [161]. If an upstream component fails health-checks, it is removed from the “routing ring”, i.e., the set of healthy upstream components, and consequently, packets are not forwarded to the unhealthy component.
• The L7 load-balancing logic typically resides in proxy servers (e.g., Proxygen [468], NGINX [382], HAProxy [4], Envoy [3]). Proxy servers or L7LBs serve as the heart of traffic management and their responsibilities go far beyond those of a typical load-balancer. Operating in different modes, they serve as reverse proxies for load-balancing, forward proxies for outbound requests (e.g., to upstream origin), and HTTP servers for handling end-user requests. The proxies terminate end-user connections and typically support multiple transport and application protocols (e.g., TCP, UDP, HTTP/1.1, HTTP/2, QUIC, TLS, publish/subscribe [357] etc.). They further contain the caching logic and leverage storage servers (e.g., memcached [344]) to store and retrieve responses for the end-user requests. For dynamic requests (i.e., requests that cannot be cached) or for requests whose cached responses are not locally available, the proxy servers redirect the requests to the application servers in the upstream origin. For relaying the requests, the edge proxies typically maintain long-lived connection with origin proxies over backbone network. L7 proxies typically use “Direct Server Return” (DSR) where the responses are sent directly to the end-users, without passing through the L4LBs [161]. DSR ensures that the L4LBs do not incur the overhead of processing return response packets, that are typically larger in size than requests [161].

• Application servers [192, 382, 6, 2, 357] serve as the eventual end-point for a request and contain the application-specific logic. Hyperscalers operate a diverse range of application servers, according to their service requirements, e.g., HTTP servers (Apache [192], NGINX [382], HHVM [7]), publish-subscribe servers (MQTT [357]), etc.

As Hyperscalers host diverse services, there is a heterogeneity with respect to the nature of connections. While most web services have typically short-lived connections between the edge and client tiers, services like publish/subscribe (pub/sub) tend to have longer connections between a client and a pub/sub back-end in the origin tier. These services use state-of-the-art pub/sub protocols (e.g., MQTT [357]) that maintains persistent connections with billions of users and require the underlying transport session between the client and the origin tier to be always available (e.g., for live notifications). Similarly, Hyperscalers also host web services like HTTP POST (e.g., video uploads), where data is uploaded from the client tier to back-end HTTP servers in the origin tier. Since these transactions mostly occur over high-RTT WAN, these services also have long-lived connections.

2.1.2 Request Workflow

Next, we provide a high-level view of a client’s interactions with the CDN, e.g., HTTP GET request for a webpage load. The web client, e.g., browser, first contacts a DNS server to obtain the IP address of the CDN’s edge proxy (phase (A) in the bottom half of Figure 2.1). It then initiates a TCP/TLS (or QUIC) connection with the edge proxy, both parties agree on the network protocols (transport and HTTP layer) for the connection, and the client sends the HTTP GET request (phase (B)). The edge proxy checks if the requested content (identified by the HTTP request URL) is available in the cache and, otherwise, requests the content from the origin server (phase (C)). Finally, the requested content is served back to the client (phase
Table 2.1: Request sub-workflows.

<table>
<thead>
<tr>
<th>Sub-workflows</th>
<th>Context</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client to edge connections</td>
<td>Between client and edge tiers</td>
<td>Inspector Gadget, Configanator</td>
</tr>
<tr>
<td>Proxy to application-server connections</td>
<td>Between edge and origin tiers</td>
<td>Zero Downtime Release</td>
</tr>
<tr>
<td>L4 to L7 packet processing and forwarding</td>
<td>Within edge or origin tier</td>
<td>Zero Downtime Release</td>
</tr>
<tr>
<td>Client-side processing</td>
<td>Within client-tier</td>
<td>WebMedic, JSCapsules</td>
</tr>
</tbody>
</table>

As an individual sub-workflow may span multiple layers and tiers, to ensure good performance and prevent request failures, the sub-workflows should be optimized not only in their local scope (i.e., within the layer or tier) but also across the layers and the tiers. For instance, an optimal-performance page load requires optimal functioning of network protocols across the different layers (e.g., TCP and HTTP) to minimize the delivery time, and optimal functioning of device software layers (e.g., browser, JavaScript V8, OS scheduler, OS memory allocators) to minimize the processing and the load time. Further restarts, due to release updates, result in disrupting a subset of sub-workflows, e.g., restarting a proxy results in breaking the connections with server and client, and, as a result, any ongoing transactions (e.g., data upload) is disrupted.

2.2 Continuous Release

Code-bases for Hyperscalers are constantly modified to introduce new functionality, bug fixes, performance optimizations, security patches and configuration changes [514, 122, 440, 240, 121]. Due to the business incentives of deploying new changes early [266, 389], Hyperscalers have adopted “continuous release” philosophy for swiftly updating code and configurations. Continuous release is defined as the “ability to get changes of all types — including new features, configuration changes, bug fixes and experiments — into production, or into the hands of users, safely and quickly in a sustainable way” [239]. Hyperscalers rely on continuous release to reduce Time-To-Market (TTM), quickly iterate to improve product quality, get early feedback from the users, and improve developer morale [239]. Recent measurements from Facebook’s CDN show that the application server tier is updated tens of times a day, with each update comprising 1000s of
code commits [122, 121, 440].

Today, the state-of-the-art approach for releasing updates requires draining connections from the applications (e.g., proxy, server) with the old code, and incrementally restarting the applications to re-initialize with the new code binaries and configurations. Hyperscalers leverage the load-balancing plane (i.e., L4LB and L7LB) in their multi-tiered infrastructure to incrementally conduct the updates. The update workflow for a proxy server is depicted in Figure 2.2. Initially at time $T_0$, all the proxies respond to L4LB health-checks to indicate their good health and receive packets for new connections. At time $T_1$, an operator decides to release update to the red instance. The corresponding instance enters a draining mode [203, 55, 52, 452] where it fails health-checks, is consequently removed from the L4LB routing ring, and receives no new connections. This phase stays active for a predefined draining period [203, 50, 51]: the time duration deemed enough for existing connections to organically conclude. Once the draining period is over, any existing connections are terminated, the instance is restarted, and the application runs with the new code and configurations (time $T_2$). At this point, the red instance starts responding to the health-checks and starts receiving new connections.

As the restarting instance receives no new connections during the draining period, the effective cluster capacity is reduced for the respective duration. To mitigate the negative consequences, such as resource contention, Hyperscalers traditionally rely on over-provisioning the deployments, or incrementally release updates in batches. Two such commonly-used techniques are blue/green deployment [451] and rolling updates [288]: the former relies on over-provisioning by maintaining two identical environments where one runs last-stable version and the other runs newer code, while the latter incrementally updates the applications in small batches. These techniques are supported by frameworks like AWS CodeDeploy, Kubernetes, NGINX, Envoy, etc.

Orthogonally, there are efforts to design update mechanisms that do not significantly harm cluster capacity. One such current state-of-the-art mechanism, called hot-restart [53], involves running multiple instances of an application on the same machine, and seamlessly migrating traffic from the old application to the newer instance with the updated code and configurations. The two application versions bind to the same ports leveraging kernel features like SO_REUSEPORT [278], the older instance enters a draining mode, while the newer instance takes-over the connection listening responsibility and manages all new connections. The kernel is able to make the distinction between the packets destined for old vs new instance through the notion of listening vs accepted socket: the application that owns the listening sockets receives the newer connections, while

Figure 2.2: Proxy update restart workflow.
packets for the existing connections are forwarded to the instance with the respective accepted sockets. Hot-restart is supported in proxy applications like HAProxy [4, 483, 323, 263], Envoy [3, 54], Proxygen [468], etc.

The traditional approaches suffer from a number of limitations. Next, we discuss the availability, performance, and operational constraints that the ideal design is required to tackle. We further provide an empirical analysis of the limitations with the traditional approaches in the next chapter (§ 3.1).

- **Availability and resource constraints:** For traditional mechanisms such as blue/green deployment and rolling updates, the restarting instances stay unavailable for the duration of draining period. Although hot-restart mitigates such unavailability concerns, it assumes that the hardware machine has enough resources to support multiple application instances — making this approach unsuitable for environments where this assumption does not hold true, e.g., application servers like HHVM are memory-constrained due to cache priming [207, 39]. The ideal scheme should be applicable to diverse environments and not incur any downtime due to its significant impact on a business’s revenue (upto $500,000 per hour for Fortune 500 companies [461]).

- **Service heterogeneity constraints:** Hyperscalers operates heterogeneous services with diverse protocols, such as TCP, UDP, HTTP, QUIC and MQTT, and different nature of connections, e.g., short vs long lived. The different services requires domain-specific draining behaviors for the connections to organically conclude. Otherwise, the connections are terminated as soon as the draining period is over and the out-of-date application is terminated. Further, the kernel abstractions used for connection migrations should be generalizable for the multiple protocols (which they are not, as we show in § 3.1). The ideal scheme should be able to prevent disruptions for the diverse services, as disruptions harm end-user QoE, trigger “re-connection storms” that put pressure on CDN/ISP networks and incur CDN-end CPU overhead to rebuild connection state [76, 36].

- **Swift release cycles:** Hyperscalers update code millions of globally-distributed servers at very fast pace, e.g., Facebook updates millions of servers atleast once every two hours [367]. Swift update cycles put pressure on the draining behavior and, consequently, on the induced disruptions as early hard restarts terminate existing connections. The ideal mechanism is required to observe the time constraints, while maintaining zero downtime, no disruptions, and incurring minimal resource overheads.

### 2.3 End-user Heterogeneity and Performance Dynamics

Due to their global-scale, Hyperscalers are expected to serve users from diverse regions, with different types of connectivity and devices. The heterogeneity in user-base presents a performance challenge due to the dependency of performance on network [464, 520, 27, 165, 365, 149, 95, 27, 383, 543, 305] and device hardware resources [20, 366, 429, 519, 371, 365]. This behavior is illustrated in Figure 2.3 as an ordered sequence of requests, with the time delays divided into network (i.e., time required by network to download an object) and computation (i.e., time required by the client to process the downloaded object)\(^1\).

\(^1\)The performance classification into network and computation is defined in earlier work [519].
Network delays: The client (e.g., browser) fist initiates a TCP and HTTP connection with the edge-server and then sends a HTTP GET request for the HTML file (foo.html). The server starts sending the bytes for the respective file, and the client receives the complete file after a network delay, which depends on a number of factors ranging from the size of file, network conditions (e.g., latency, bandwidth, packet loss), to server’s network stack configurations (e.g., TCP and HTTP). Server’s network stack — our focus for improving network performance — plays a key role in determining the network delay as it controls the flow of data, i.e., the number of bytes that can be sent at a point in time. Servers typically use transport protocols, such as TCP congestion controls (CC), to dictate the data transmission. The primary goals of TCP-CC are to ensure reliability of data transfer and maximize the network usage in a fair manner. A number of CC algorithms have been designed that model different network features (e.g., latency, loss rates, bottleneck buffers) to tame data transmission. TCP further include a number of configurations that control various aspects of CC, e.g., initial congestion window to control the number of packets sent in the first round-trip, retransmission timeout to decide when a packet is considered to be lost and should retransmitted, etc. Similarly, HTTP protocol and its configurations dictate rules for sending requests and receiving responses between the clients and servers. Currently, three HTTP versions (1.1., 2.0 and 3.0) are used in-the-wild, they differ widely in their designs, ranging from their use of transport to request/response mechanics. While HTTP over TCP is the current de facto standard for most web traffic, Hyperscalers also employ other transport protocols such as UDP, cross-layer protocols such as QUIC, and specialized protocols such as publish/subscribe MQTT, for different services.

Computation delays: Once the client has received the HTML file, it incurs a computation delay for parsing the file, processing the code (e.g., JavaScript or CSS defined in HTML), and rendering the HTML file on device screen. The HTML defines a static Document Object Model (DOM) [336], which is an object-oriented representation of a webpage containing its structure, content, and styles. Programming Languages
like JavaScript can interact with DOM to modify the internal structure. The browser first parses the HTML, CSS, and JS to create and augment the DOM and further creates a CSSOM tree (containing CSS-defined styles for the DOM nodes). It next combines DOM and CSSOM to generate a render tree which serves as the basis for determining the layout of content and painting the content on screen. Any change in DOM or CSS (e.g., due to a JS API call) can have a cascading effect leading to re-generating the render tree, recalculating the layout, and appropriately generating new frames to paint on the screen. Device’s hardware resources play a key part in determining the computation delay, as code processing requires computational cycles and memory resources. Consequently, a device with superior hardware resources (e.g., a desktop) is expected to incur lower computation delay as compared to one with constrained resources (e.g., a low-end phone). While processing the HTML and JavaScript, the browser may send out more GET requests for fetching referenced objects (e.g., images, CSS, video), and incurs further network and computation delays.

Device memory — our key focus for client-side resource — plays a critical role, as the browser requires memory for storing DOM data structures, media content, evaluating and processing JS code, and rendering the web objects on screen. Memory contention, especially for low-end phones with modest resources, has a direct impact on the performance, e.g., bloated computation delays due to application stalls for garbage collection. In the worst case, memory contention can crash the browser, making the website unavailable to the end-user. Consequently, browser developers have paid special attention to memory management systems, and modern browsers, such as Chromium-based Brave, Edge, Chrome, etc., use a number of memory allocators for managing memory. Table 6.2 lists these allocators and summarizes their purpose during the different phases of page load process. The browser leverages these allocators to allocate memory during different stages of the page load process, e.g., it may allocate memory to store media content (e.g., media for images), execute JavaScript code and store JavaScript objects (e.g., v8 for code, strings, arrays), and render the DOM on screen (e.g., malloc for DOM data structures, cc for rendering). Additionally, JavaScript API calls may trigger a series of cascading events, each with their own memory implications, e.g., adding a node to the DOM through JavaScript results in the memory allocations for recalculating the layouts, generating the render state, and painting the content. Consequently, a complete visibility into the website code, browser mechanics, and allocators is required for a holistic understanding of the source of memory overhead and its performance implications.

Improving page load performance: In the context of growing heterogeneity in the user-base, it is challenging for the Hyperscalers to democratically reduce the network and computation delays for the diverse connections and devices. While related works have proposed a number of optimizations, such as (i) edge optimizations (e.g., minimizing latency [101, 58, 237], resource contention [553], caching [459, 65]), (ii) protocol optimizations (e.g., TCP [383, 187, 520, 95, 19], HTTP [267, 557, 520, 165, 27]), (iii) reducing network overhead [173, 167, 308, 276, 106, 167], (iv) inspection of webpage complexity and dependencies to improve prioritization and scheduling [88, 331, 309, 375, 522, 171], (v) understanding device-side resource limitations [20, 429, 518, 371, 123], to (vi) reducing website code complexity [168, 292, 509, 147], the focus of this thesis is on improving performance by tuning protocol configurations and gaining visibility into the client-side memory dynamics of the page load. In this context, the ideal designs are required to address the

---

2We use the different phases during the page load process as defined in Chrome standards [140].
following objectives and limitations, further empirically analyzed in the next chapter (§ 3.2).

- **Protocol performance sensitivity:** Protocol performance varies across networks, e.g., TCP-CC is highly sensitive to network path [543], HTTP performance varies based on network conditions and website complexity [520]. Table 5.1 lists the set of configurations for L4 and L7, as well as their setting parameters. To achieve optimal performance, Hyperscalers are required to cater for the unique network characteristics of individual user connections, e.g., performance optimizations geared towards one class of network may not scale/apply to the other classes [464, 365, 383]. However, designing such optimizations is challenging as it requires modeling techniques and algorithms to model the performance sensitivity of protocols and efficient network stacks for configuring protocol configurations on a per-connection level. In § 3.2.1, we empirically discuss the limitations with traditional tuning approaches, and discuss the need for cross-layer configuration tuning for optimal web performance.

- **Device hardware dynamics:** The device hardware memory constraints can be addressed from a number of fronts. One can design low-weight OSes, optimized browsers or JavaScript engines, or specialized versions of websites for the diverse devices. However, the ideal approach should be practical and readily deployed in today’s infrastructure, e.g., while designing light-weight OSes is a promising approach, such approaches showed limited benefits in the past due to deployment issues (network costs hindering the download of updates [173, 387] and out-of-date device configurations complicating the application of updates). Designing optimized browsers or websites first requires efficient visibility techniques for understanding the dynamics between website code and resource usage. However, such visibility techniques should be holistic in identifying the resource interactions between the different device layers, e.g., hardware layer, OS, browser and the website code. In § 3.2.2, we discuss the limitations with traditional visibility approaches, and discuss the need for a cross-layer memory analysis technique for accurately understanding the memory implications of today’s web.
Chapter 3

Motivation

In this chapter, we discuss how the limitations in the traditional designs prohibit the effectiveness of Hyperscalers to meet the demands of growing digital services (i.e., frequent code updates and diverse user-base). We discuss the challenges faced towards ensuring zero-downtime and high-performance of services, and how cross-layer solutions can help in improving QoE.

3.1 Ensuring Zero-Downtime in the Era of Continuous Release

While the high-level request workflow from Figure 2.1 may appear straight-forward, each end-user interaction with the CDN requires interactions across multiple layers within or across tiers. Importantly, to operate the request workflow in a well-oiled manner, the individual interactions may need to be optimized not only in their local scope, but across the different layers involved in the workflow. However, unfortunately, the traditional approaches for continuous release adopt simplicity in favor of the additional cost that may be required for cross-layer optimizations. Consequently, they lead to disruption for diverse services, leading to inconsistencies in sub-workflows discussed in Table 2.1. Next, we focus on three such classes of disruptions observed at a Hyperscalers and discuss how cross-layer mechanisms can mitigate the disruptions without adding significant additional cost.

**Disruptions in QUIC services:** Services based on QUIC, such as HTTP/3, webRTC, etc., use connectionless UDP for transport and typically create a pre-defined number of UDP sockets for sending and receiving the packets. The service application directly processes the UDP packets and is responsible for maintaining the connection state. Their architecture is strikingly different from TCP-based services, that maintain listening sockets for creating new connections and accepted sockets — one per connection — for managing the state, e.g., application that holds the reference to a certain accepted socket gets forwarded (from the kernel) the packets for the respective connection. The cross-layer nature of QUIC services presents a problem, as the traditional update release mechanisms like hot-restart only operate at the transport layer level: the updated instance take over the listening sockets, while the older instances keep managing the existing connections as it holds the accepted sockets. Due to the connection-less nature, UDP uses a single set of sockets for
managing packets for both new and existing connection, and taking over the UDP sockets, similar to TCP hot-restart, leads to significant consistency issues as all the packets, whether belonging to the existing or the new connections, are forwarded to the updated instance. Since the updated instance does not hold the connection state for the packets, it terminates the connections.

To highlight this problem, we deployed hot-restart at 10 Meta edge clusters, and present the median number of mis-routed UDP packets per proxy instance over the course of 20 minutes draining in Figure 4.5. We observed a sharp increase in the number of mis-routed packets as soon as the restart initiated: tens of thousands of packets are wrongly forwarded to the instance that lacks the connection state. The number of mis-routings gradually decline as the earlier mis-routed packets illicit connection termination and, effectively, the total number of UDP connections for the instance declines over time. These disruptions are observed by end-users in form of connection terminations and timeouts.

At its core, the connection disruptions triggered by the out-of-order packets are due to a lack of flexibility in the single-layer-oriented connection primitives. Being limited to only the transport layer, such primitives, e.g., SO_REUSEPORT [278] or SCM_RIGHTS [325] used for hot-restart, are unable to address the requirements of cross-layer protocols like QUIC and break the consistency for the L4-to-L7 packet processing sub-workflows (Table 2.1). In fact, the existing designs can be conservatively extended to add support for QUIC without the need of large-scale connection state migration across the application instances. Assuming we have a cross-layer solution that span the transport and application layers, we can ensure the consistency
of packet forwarding by adding an additional data-path between the draining and the updated instance. Using such a data-path, packet delivery across the layers can be coordinated to ensure that packets always make their way to the instance with the corresponding connection state.

**Disruptions in publish/subscribe services:** Services such as publish/subscribe (pub/sub) use specialized protocols that maintain persistent, long-lived connections between the client and back-ends. MQTT [357] is one such protocol that is currently used for planet-scale pub/sub services, e.g., Facebook Messenger [550]. The pub/sub clients are directly connected to a pub/sub back-end in the origin-tier, and packets between the two end-points are relayed by the intermediary L7LBs in the edge and the origin tiers. Due to their peculiar, long-lived connection nature, they are disrupted by continuous release: the connections often outlive the draining period (10-20 minutes at Meta) when a L7LB on the connection path restarts and, consequently, are terminated once the instance hard restarts at the end of draining period. These terminations are highly disruptive to the end-users (lead to service downtime) and the CDN itself (trigger reconnect storms that incur network and CPU overhead [76, 36]).

To highlight this problem, we deployed hot-restart at 10 Meta origin clusters, and monitored the disruptions for the MQTT connections. Figure 4.14 presents the results, where T=20 indicates the beginning of L7LB updates (10% of origin L7LBs restarted) and the vertical line indicates the end of draining period (L7LBs in origin-tier at Facebook are configured to use a 10 minute draining period). We quantify the number of MQTT connection disruptions by measuring the number of ACKs received, representing the client reconnects triggered by abrupt terminations. As soon as the draining period ends and the updating L7LBs are hard-restarted, the connections between MQTT back-ends and clients are terminated. The terminations trigger the client applications to setup new connections to maintain their persistent connectivity, leading to a sharp increase in the number of connection retries.

The key root-cause of the of these disruptions is that the traditional mechanisms work in the context of individual layer/tier, while pub/sub connection span multiple layers across tiers. Hot-restart based solutions that operates in a local context (i.e., within a machine) cannot prevent this disruption as a cross-tier connection coordination is required to mend the transport between the restarting L7LB and its upstream/downstream. Further, the traditional mechanisms rigidly apply same draining rules for diverse connections (i.e., short vs long lived). While an operator can inflate the draining periods to cater for pub/sub connections, this approach is highly sub-optimal due to its downtime and resource overheads: leading to either reduced cluster capacity as draining instance does not manage new connections, or high overheads of running two proxy instances on same hardware for an extended period. The key towards designing a solution that adheres to the downtime and operational constraints is to develop protocol abstractions through which the downstream and the upstream tiers can coordinate to migrate the data-path from the restarting L7LB to another healthy L7LB, thereby making the restart transparent to the clients and back-ends. The transparent nature of the restart with respect to the end-points preserves the state at the respective ends, and only a minor communication cost is required to materialize such a migration of connection among the intermediaries. However, due to the unavailability of such abstractions, such migrations are challenging to implement.

**Disruptions in HTTP Uploads:** Services such as HTTP POST upload manage data uploads from the end-users to the back-end HTTP servers, typically over high-RTT WAN. Due to operational constraints,
such services have very short draining budgets, e.g., 10-15s for as compared to 10-20 minutes for L7LB at Meta. Consequently, the uploads outlive the draining period and are terminated when the application server is restarted. These terminations are highly disruptive to the end-users, as request state (i.e., data uploaded yet) is lost, and requires the end-user to send the data again from scratch. These disruptions not only degrade the end-user QoE but may also lead to wastage of data plans. Note that, these disruptions are fundamentally different from pub/sub connections, as the connection end-point, i.e., a webservice, is restarted, instead of an intermediary restart. Due to this fundamental difference, the disruption cannot be maneuvered by simply migrating the request to another healthy component, as the restarting server holds the request state and loses the state once restart is initiated.

To highlight this problem, we measured the number of upload disruptions at the entire Meta’s web tier for 7 days. Figure 3.3 plots the percentage of POST failures during restarts (HTTP 500 errors sent to end-users). Note that, the application servers are restarted tens of times a day and the 7 days worth of data covers around 70 web tier restarts. Although the percentage might seem very small (e.g., 0.0008 at median), there are billions of POST request per minute for the entire web-tier and even the small percentages translate to huge number of requests (e.g, ~6.8 million for median).

The key root-cause of these disruptions is the rigidity in the load-balancing plane (i.e., L7LBs) of committing to a specific back-end server to complete a request: the upstream HTTP back-end once selected by the downstream L7LB is expected to fulfill the request in completion. While this design choice suits the primary workload of HTTP back-ends (i.e., short-lived API requests), it is unsuitable for long-lived requests like POST uploads. Intuitively, the request state (i.e., the data uploaded until server is restarted) can be stored at the origin-tier and migrated to another healthy server instance, without the overhead of client re-sending the data. However, such mechanisms require coordination between servers and their downstream L7LBs and abstractions in HTTP protocol through with the migration can be realized. Lacking such abstractions, the server has no option but to respond with HTTP 500 (i.e., error code) and requires the end-user to upload the data again from scratch.

### 3.2 Ensuring High Web Performance for the Heterogeneous User-base

Web performance is impacted by a number of factors, ranging from network, client-device to website complexity. To improve performance, CDNs are required to optimize the delivery of web objects (i.e., interactions between the edge and the client tiers), and optimize the loading of web objects on the client device (i.e., interactions among the client layers such as browser, JavaScript engine, OS, hardware resources). As Hyperscalers serve diverse user-base with heterogeneous connectivity and devices, the ideal design should be able to cater for the unique characteristics of the user-base, e.g., use the protocols that are more suited for the user network to minimize delivery time, design websites that do not overwhelm a low-end device hardware resources. Next, we discuss the limitations in traditional CDN design that makes the ideal design challenging, and discuss role of cross-layer flexibility to improve performance in a democratized fashion.
3.2.1 Protocol Configuration Tuning at Edge

Improving network delivery performance is critical towards improving QoE of websites. Network protocols, such as TCP and HTTP, dictate the transfer of bytes from the edge to the end-users, and, consequently, have a direct impact on performance. However, due to user-base diversity, Hyperscalers face a challenge regarding the selection of right protocols and configuration from the edge: the optimal choice is contingent on the network conditions [520, 165, 464, 27, 383, 543, 305], end-user devices [20, 429, 371], and website complexity [85, 88, 375, 521, 519]. Many measurement studies [520, 165, 202, 27, 152] have explored the performance sensitivity of different protocols and configurations, and a number of systems have been proposed that aim to tune a subset of network stack with the optimal configurations based on the end-user network or device. These optimizations range from tuning TCP initial congestion windows [383, 187], congestion controls [529, 442, 149, 251, 305], to HTTP configurations [267, 557].

CDNs today either employ a static or dynamic approach towards selecting their network stack configurations. The static approach either simply uses the default configurations (i.e., configurations set by default in OSes and proxy/server applications), or measurement-driven configuration hand-tuning. For the latter case, operators actively evaluate the performance implications of various configurations in-the-wild, and use these measurements as the basis for discovering the best-performing configurations. Some examples include regional latency measurements by Cloudflare [213, 412], TCP congestion control measurements from Verizon EdgeCast [456] and Facebook [199], TCP Initial Congestion Window (ICW) experiments at Yahoo! [27], HTTP configuration experimentation at Cloudflare [485, 341], and others [209, 279]. The top-performing configurations are either used homogeneously for the entire user-base, or certain coarse-grained classes of connections (e.g., cellular traffic), workloads (e.g., video traffic) or devices (e.g., mobile phones).

In the dynamic approach, CDNs build data-driven systems that leverage active measurements and dynamically tune the configurations on an AS, prefix, or workload basis, e.g., ICW tuning for inter-DC traffic at a PoP-granularity at Verizon EdgeCast [187], ICW tuning at a granularity of end-user clusters (/24 IP prefix) at Baidu [383], congestion control tuning (switching between Cubic and BBR) at a granularity of end-user clusters (/24 IP prefix) at Verizon EdgeCast [37], etc. Similarly, Akamai also reportedly tunes congestion control (switching between Fast and Cubic) at a connection granularity [23, 22]. However, the configuration space is mostly limited to individual layers and a narrow subset of available configurations, e.g., ICW in [383, 187], two variants of congestion control in [37, 22], a single HTTP configuration in [267].

The traditional approaches face three key challenges. First, static approaches do not account for the heterogeneity in user connections, thereby a dynamic approach towards configuration tuning is required. However, such dynamic techniques require domain-specific learning algorithms to account for the diversity and learn the optimal configurations in a principled manner. The design of such algorithms is challenging due to the high volatility in network conditions, high-dimensionality due to device/diversity, and the cost associated with the QoE of services (i.e., bad configurations hurt QoE and revenue). Second, traditional approaches limit the tuning to either single knob or layer. However, as different layers in the network stack (e.g., transport and application) directly impact the performance, cross-layer tuning is expected to bring substantially more benefits. Third, tuning cross-layers configurations in a fine-grained manner (e.g., on a per-connection basis) is challenging due to a limited flexibility in the system design of networking stacks and
web server applications, e.g., some parameters (e.g., ICW) can be configured on the connection level, while others can only be configured on a global scale (e.g., tcp_low_latency). Tuning at a coarser granularity (e.g., global scale variables in kernel), either limits the type of supported connections on a machine or limits the configuration space.

To evaluate the limitations with traditional approaches, we quantified the benefits of tuning networking stack by conducting a large scale study in a local test-bed. We emulated a wide range of representative networks (extracted from a production CDN trace) and perform an exhaustive, brute-force search of configuration space (TCP and HTTP configurations). In each trial, the server iteratively selects a configuration from the possible configuration space, a representative network is emulated using NetEM [223], and the PLT of a randomly selected website from Alexa Top-100 (locally cloned on the server) is measured five times. We further leveraged our configuration inference tool for inferring the configurations that current CDNs employ at their edge (referred to as “HandPicked”).

Figure 3.4 compares the improvement in page load time (PLT) for the different schemes versus the Linux/Apache default settings. Leveraging the brute-force evaluation of configuration space, “Oracle” always selects the best-possible configuration for a connection and puts an upper bound in performance improvements. We observed that while “HandPicked” outperforms the default at median, it lags far behind the oracle (up to 4X better improvement at tail). These results show the limitations of the status-quo tuning strategy and highlights the possible potential of network-aware, data-driven tuning for configuring the right network protocol settings at the edge. We further evaluated the case where a single layer is dynamically tuned, as done for other systems [383, 187, 37]. “TCP_CC” lines plots the results for such a system that leverages the oracle to tune TCP congestion control. We observed that cross-layer tuning results in significant more benefits, e.g., tuning only CC improves PLT by less than 8% at the median, highlighting the clear benefits of cross-layer tuning. We observed even lower improvements for similarly tuning TCP ICW or HTTP layer configurations. Finally, we evaluated “Bayesian Optimization” (BO), a state-of-the-art statistical technique used for tuning systems configurations [29, 300, 151, 504]. BO is a principled global, black-box optimization strategy that uses prior probability functions to capture the relationship between the objective function (e.g., performance of a configuration) and the observed data samples. BO models the objective function as a distribution of
candidate Gaussian processes [287], and the candidate functions are used to select the next promising point (i.e., configuration) which is then evaluated on a connection. BO then updates its posterior belief by adding the new observation to the set of seen observations and the prior gets consolidated with the new evidence. While fine-grained, this approach is relatively static and does not re-evaluate old choices, and is thus unable to adapt to network dynamics [310]. We observe the effects of this rigid behavior with significantly poor performance as compared to oracle, motivating the design of domain-specific algorithms.

The control over the edge networking stack and web application, as well as visibility into the website performance, presents a unique opportunity for the Hyperscalers to design feedback/data driven systems for dynamically tuning the cross-layer networking configurations. Provided that efficient algorithms and flexible system techniques are available for tuning connections in a fine-grained manner, the control can be leveraged to build systems that continuously monitor the performance impacts of edge network configurations, and dynamically tune the appropriate configurations that are suited to the characteristics of individual connections.

### 3.2.2 JavaScript Memory Overheads at the Client-side

To satisfy end-users’ requirements, websites are constantly updated to add more functionality and, consequently, their complexity and resource usage has steadily climbed over time. We observe that the memory requirements of modern websites have steeply climbed over the years, with our measurements of Alexa top 250 websites over the course of 2015 to 2021 showing that the memory requirement of the median website has increased by 40% (more than doubled for the tail website). More importantly, our measurements show that JS is the key dominant factor behind web’s memory overhead, contributing 64.8% of memory for the median website in 2021. In contrast, the hardware memory in mobile devices has not grown at a similar rate, e.g., the median device in developing regions has the same memory in 2019 as it had in 2015 [366]. This presents a significant performance challenge as memory constraints in mobile devices, especially low-end ones, have a direct impact on both the performance (e.g., due additional computational overhead of garbage collection) and usability of web pages (e.g., browser crashes).

A key challenge towards understanding JS memory is that JS often leads to cross-layer (or component) interactions, leading to direct (i.e., V8) and indirect (i.e., components outside V8 boundary) memory allocations, thus requiring nuanced visibility into the JS and browser mechanics to identify the source of overhead. To illustrate, let us consider a code example (Listing 6.1) where a function `bar()` uses jQuery to iterate over boxes in HTML and modify their text and style. Several browser components and allocators are involved during `bar()`’s execution and the listing annotates the scope of JS interactions and list the related browser events triggered by the JS code line. While parsing and compiling the code, V8 emits events, such as `compile` and `FunctionCall`, and the memory overhead of such tasks is dynamic, depending on the bytes of code, size of included library, types of APIs, types and size of runtime objects defined (e.g., `boxes` array), etc. When the jQuery APIs (`text` and `width`) interact with DOM to modify text node and style, rendering engine emits

---

1. We leveraged the WayBack Machine from the Internet Archive [44], and measured memory requirements for the historical snapshots of the respective websites from 2015 to 2021. For each website, we capture snapshots from each quarter of the year, randomly select one snapshot per-quarter, and repeat the measurement ten times. This process is repeated for every year, from 2015 to 2021.
events, such as `InvalidateLayout`, `ScheduleStyleRecalculation` and `UpdateLayoutTree`, the compositor allocates memory to update layout and generate the tiles on screen, and the memory overhead is variable with respect to the type of content added and the complexity of DOM changes. Similarly, if any network fetches were involved, the overhead of storing the objects would highly depend on the type and size of the respective objects.

Further, while loading a webpage, the browser executes thousands of JS functions (~2800 at median for Alexa top 1K websites) and emits tens of thousands of fine-granular browser events. In Figure 6.3, we plot the duration between two consecutive events for top 1K websites and the 10s of millisecond duration at median shows the fine-granularity of browser events. In fact, a majority of such events (>70% for median top 1K websites) are precipitated by JS and are outside scripting (i.e., V8) scope: Figure 6.4 plots the percentage of categorized events triggered by JS and we observe significant percentage of layout, painting, loading and layer events. These trends present two challenges: first, fine-granular memory measurements are required to account for the cross-component allocations made in response to executing a JS functions, and, second, specialized techniques to attribute events to JS functions are required to attribute the direct and indirect root-causes.

Unfortunately, the existing techniques — OS-level [217, 154, 405] and Browser-provided [491, 137, 11] tools — fail in meeting these criteria. Timeline Profile and HeapProfiler are based on Chromium: the former captures a time-series of JS heap and browser activity (e.g., network, rendering, scripting events), while the latter captures V8 heap memory broken down by JS functions and objects (e.g., strings, arrays). We observe low coverage (below 30% at median) for both, the key reason being that these tools are only limited to V8 heap and do not capture the indirect JS memory. In Figure 6.6, we plot the percentage of JS memory contributed by the various allocators and observe that a variety of non-V8 allocators, such as `cc` or compositor, `malloc`, etc., make significant contributions, with V8 contribution at only 27% at median. Our conversation with the Chrome developer indicated that although the report generated by HeapProfiler includes DOM node references, the tool itself does not measure the memory allocated to objects being referenced and, thus, only captures a subset of the overall memory footprint.

On the other hand, MemoryInfra and OS-based tools provide full coverage: the former resides inside the browser and captures all allocators, while the latter (e.g., `meminfo` [217], `dumpsys` [154] and `perfetto` [405]) capture information at the process level and provides an aggregated memory view. Although these tools do not natively measure JS memory, recent work [12] used such tools to capture JS memory by removing all JS code from a website and taking a difference between vanilla website’s memory (i.e., with all the JS code intact) and the stripped-off version. While these tools provide good coverage, they are limited in their ability to localize the source to specific functions and identify the root-cause of the overhead. Since OS-based tools capture information at the process level, they do not have any visibility into the functions being processed and the corresponding browser mechanics, and thus provide coarse-grained information. Similarly, MemoryInfra measures memory at a 10s granularity by default and only provides an after-the-fact memory snapshot, i.e., total allocator-wise breakdown of memory consumed due to the browser work in the preceding 10 seconds. Though one can modify the MemoryInfra code to increase the frequency, its major deficiency is that it only localizes the overhead to coarse-grained allocators and does not capture fine-grained memory for individual
functions. Further, it lacks the ability to connect the cascading events to the source function (i.e., root-cause events).

Thus, while the existing tools are able to identify the symptoms, i.e., how much memory is allocated?, they fail in attributing the source of this symptom due to their coarse-granularity and lack of event attribution to functions. Consequently, they do not provide enough information to the developers to diagnose the key JS functions that contribute the memory and identify the key browser mechanics that result in the overhead. From a developer’s perspective, this context is essential for identifying what code or part of website design to fix and, in its absence, the space of optimizations is vast and arbitrary, requiring high cost from the developer end to find and optimize the problems.
Chapter 4

Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website

4.1 Introduction

Online service providers (OSP), e.g., Facebook, Google, Amazon, deploy massively complex code-bases on large sprawling infrastructures to deliver rich web services to billions of users at a high quality of experience. These code-bases are constantly being modified to introduce bug fixes, performance optimizations, security patches, new functionality, amongst a host of other reasons. Recent studies from Facebook [440, 427] show that each day tens of thousands of commits are pushed to tens of thousands of machines across the globe. In fact, the number of web-tier releases increased from once per week in 2007 to tens of times a day in 2016, each comprising of 1000s of code commits [122, 121].

At the scale of multi billion users and millions of machines, code-update and release techniques must be swift while simultaneously incurring zero downtime. Today, the state of the art approach for deploying these code changes requires draining connections from servers with the old code and incrementally restarting the servers to introduce the new code [55, 203, 452]. This technique can have a host of undesirable consequences from lowering aggregate server capacity and incurring CPU overheads to disrupting and degrading end user experience. At the scale of billions of connections, restarting connections is disastrous for the ISP, end-user, and the OSP [76, 36]. The process of connection-restart incurs a number of handshakes (e.g., TLS and TCP) which we show (in Section 4.2.5) consumes as much as 20% of the OSP’s CPU. Additionally, the flood of new connections triggers wireless protocol signaling at the cellular base-station which simultaneously drains a mobile phone’s batteries and can overwhelm the cellular provider’s infrastructure. Finally, we observed that during the restarts, users can experience QoE degradation and disruptions in the form of errors (e.g., HTTP 500 error) and slower page loads times (i.e., due to retries over high-RTT WAN).
Motivated by the high code volatility and the potential disruption arising from code deployment, many online service providers have turned their attention to designing practical and light-weight approaches for transparently deploying code in a disruption free manner, i.e., deploying code while ensuring zero downtime. The design of such a disruption free update mechanism for large providers is challenge by the following characteristics which are unique to the scale at which we operate: first, large providers employ a large range of protocols and services, thus, the update mechanisms must be general and robust to different services. For example, we run services over both HTTP and MQTT (a publish-subscribe protocol [357]) which have distinctly different tolerance and state requirements. Second, while many applications are stateless, a non-trivial set of applications are stateful, thus the update mechanisms must be able to seamlessly migrate or transparently recreate this state at the new server. For example, a non-trivial number of connection are long-lived and often transfer large objects, failing to migrate state can significantly impact end-user performance. Third, due to energy reasons and application-specific requirements, a subset of servers are resource-constrained (e.g., cache priming [207, 39] for HHVM servers [6] consumes most available memory), which prevents us from running both the new code and old code on the same server, thus preventing us from leveraging kernel mechanisms to migrate connections.

Our framework builds on several unique characteristics shared by the infrastructure of online service providers such as Facebook and Google. First, the end-to-end infrastructure is owned and administered by the provider which implies that updates in a specific tier, can leverage a downstream tier to shield the end-user from disruptions, e.g., the application tier can leverage the reverse proxy tier. Second, while application state is hard to extract and isolate, for long lived connections with large objects, we can recreate the state by identifying the partial requests at the old server and replaying them to the new server. Together these insights allow us to transparently migrate from old to new code while restarting servers without exposing the loss of state or server termination to the end-user.

In our framework, an update is achieved by signaling the upstream tier to handle connection migration and by indirectly externalizing user-specific state and redirecting the state to an upstream tier. The upstream tier redirects existing connections and the externalizes the state to the new servers (or servers with the new code). Additionally, zero downtime for a restarting L7LB is achieved by handing-over traffic to side-car (instance with the new code) on the same machine. To this end our framework consists of two mechanisms: a technique for signaling and orchestrating connection hand-off via an upstream tier, a method for detecting and externalizing state, and enhancements to pre-existing hand-off kernel-based mechanisms to enable them to scale to billions of users.

This framework has been deployed at Meta for several years and has helped to sustain an aggressive release schedule on a daily basis. While comparing our framework to previously used release methodologies, we observed that our framework provided the following benefits: (i) we reduced the release times to 25 and 90 minutes, for the App. Server tier and the L7LB tiers respectively, (ii) we were able to increase the effective L7LB CPU capacity by 15-20% , and (iii) prevent millions of error codes from being propagated to the end-user.
### 4.2 Background and Motivation

In this section, we introduce Meta’s end-to-end web serving infrastructure and present motivational measurements from the production clusters.

#### 4.2.1 Traffic Infrastructure

Figure 4.1 provides an overview of Meta’s geographically distributed multi-tiered traffic infrastructure, comprising of DataCenter (order of tens) and Edge PoPs (Point-of-Presence, order of hundreds). At each infrastructure tier, Meta uses software load-balancers (LB) to efficiently handle diverse user workload requirements.
and QoE. Additionally, at the origin data centers, there are also application servers in addition to the LBs.

- **L4LB (Layer4LB):** Meta uses Katran, a transparent, XDP-based [9], L4LB layer that serves as a bridge in between the network routers and L7LB (proxies). Routers use ECMP [232] to evenly distribute packets across the L4LB layer, which in turn uses consistent hashing [9, 161] to load-balance across the fleet of L7LBs.

- **L7LB (Layer7LB):** For L7 load-balancing, Meta uses Proxygen, an in-house proxy with responsibilities encompassing beyond those a typical traditional L7LB shoulders. Operating in different modes, it serves as the reverse proxy for load-balancing, forward proxy for outbound requests and HTTP server. Proxygen is the heart of traffic management at Meta, supporting multiple transport protocols (TCP, UDP), application protocols (HTTP/1.1, HTTP/2, QUIC, publish/subscribe [357] etc.), serving cached content for CDN, maintaining security (TLS etc.), health-checking and monitoring upstream app. servers etc.

- **App. Server tier:** Resides in the DataCenter and ranges from web (HHVM, django, custom apps. built leveraging the Proxygen HTTP server library [6, 2, 468]) to special-purpose servers (e.g., Publish/Subscribe brokers [357]). HHVM servers (our focus in application tier) are a general purpose application server for HACK [470], with workloads dominated by short-lived API requests. However, they also service long-lived workloads (e.g., HTTP POST uploads).

### 4.2.2 Life of a Request

In this section we present a detailed view of how user requests are processed by the various tiers of our infrastructure and in doing so we highlight different application workflows and how they are treated.

1. **Edge PoP** serves as the gateway into our infrastructure for a user’s request and connections (TCP and TLS). These user requests/connections are terminated by the **Edge Proxygen**.

2. **Edge Proxygen** processes each request and, if the request cannot be serviced at the **Edge**, it forwards the request to the upstream **Origin DataCenter**. Otherwise, for cache-able content (e.g., web, videos etc.) it responds to the user using **Direct Server Return** [9].

3. **Edge** and **Origin** maintains long-lived HTTP/2 connections over which user requests and MQTT connections are forwarded.

4. **Origin Proxygen** forwards the request to the corresponding **App. Server** based on the request’s context (e.g., web requests to **HHVM**, django servers while persistent pub/sub connections to their respective MQTT broker back-ends).

In this paper, we focus on restarts of Proxygen (at Edge and Origin) and HHVM App. Server (at DataCenter), and focus on the traffic for cache-able, uncache-able, and MQTT-backed content. Our goal is to design a framework that shields transport (TCP and UDP) and application protocols (HTTP and MQTT) from disruption, while still maintaining reasonable update-speeds and zero downtime.

### 4.2.3 Release Updates

Traditionally, operators rely on over-provisioning the deployments and incrementally release updates to subset of machines in batches. Each restarting instance enters a **draining mode** during which it receives no new
connections (by failing health-checks from Katran to remove the instance from the routing ring). This phase stays active for the **draining period** [203, 50, 51], the time duration deemed enough for existing connections to organically terminate. Once draining period concludes, the existing connections are terminated and the instance is restarted and the new code kicks-in.

### 4.2.4 Motivating Frequent Restarts

To swiftly address security concerns and adapt to evolving user expectations, frequent code releases have become the norm [427, 122, 440, 240], not the exception. In Figure 4.2, we present the number of global roll-outs per week, over a period of 3 months for 10 Meta’s Edge and DataCenter clusters.

**L7LB:** Globally, at the L7LB tier, we observe on average three or more releases per week. In Figure 4.3, we analyze the root-cause of these releases and observe that the dominants factors are binary (i.e., code) and configuration updates. We note that unlike other organizations, where configuration changes might not necessitate a release, Meta requires restarting the instances for configuration update. This is an artifact of system design and, since Zero Downtime Release-powered restarts do not result in disruptions, it removes the complexity of maintaining different code paths, i.e., one for robust restarts for binary updates and another for configuration-related changes. Binary updates (due to code changes) always necessitate a restart and account for ∼47% of the releases, translating to multiple times a week.

**App. Server:** At the App. Server tier (Figure 4.2), we observe that, at the median, updates are released as frequently as 100 times a week [427, 440]. We also observed that each update contains anywhere from 10 to 1000 distinct code commits (Figure 4.4) and such high degree of code evolution necessitates frequent restarts. Conversations with Meta developers identified that the constant code changes to the app. tier is a function a cultural adoption of the “Continuous Release” philosophy. Although the App. Server tier evolves at a much higher frequency, the impact of their restarts can be mitigated as L7LBs terminate user connections and can shield the users from the App. Server restarts. However, due to the stateful nature of the App. Server tier and the high frequency of code updates (and thus restarts), some users are bound to receive errors codes (e.g., HTTP 500) and timeouts.

### 4.2.5 Implications of Restarts

Although the benefits of constant and frequent releases are obvious, we observed that at our scale the implications and consequences can be subtle and far-reaching. The “disruption” induced by a release is measured along multiple dimensions, ranging from increase in resource usage at CSP-end to a higher number of failed user requests. Specifically at Meta, any irregular increase in the number of HTTP errors (e.g., 500 response code), proxy errors (e.g., timeouts), connection terminations (e.g., TCP RSTs) and QoE degradation (e.g., increased tail latency) quantify the extent of release-related disruptions. Next, we elaborate on the direct and indirect consequences of a release.

- **Reduced Cluster Capacity:** Intuitively, during a rolling update, servers with old code will stop serving traffic and this will reduce the cluster’s effective capacity. An unexpected consequence of reduced capacity is increased contention and higher tail latencies. To illustrate this point, in Figure 4.6, we plot the capacity for an
Edge cluster during a release. From this figure, we observe that during the update, the cluster is persistently at less than 85% capacity which corresponds to the rolling update batches which are either 15% or 20% of the total number of machines. Minutes 57 and 80-83 correspond to time gap when one batch finished and the other started. In a complementary experiment, we analyzed the tail latency and observed significant increase due to a 10% reduced cluster capacity.

- **Increased CPU Utilization:** During a server restart, the application and protocol states maintained by the server will be lost. Clients reconnecting to Meta’s infrastructure will need to renegotiate this application and protocol state (e.g., TCP, TLS state). In Figure 4.7, we plot the CPU utilization at the App. Server-tier when clients reconnect. We observed that when 10% of Origin Proxygen restart, the app. cluster uses 20% of CPU cycles to rebuild state. This overhead mirrors observations made by other online service providers [76, 36].

- **Disrupted ISP Operations:** At the scale of our deployment, a restart of billions of connections can also put stress on the underlying ISP networking infrastructure, especially the cellular towers. On multiple instances, ISPs have explicitly complained about the abrupt re-connection behavior and the resulting congestion on the last-mile cellular connections.

- **Disrupted end-user quality of experience:** Despite efforts to drain connections, restarts at the Proxygen or App. Server tiers lead to disruptions for users with long-lived connections (e.g., MQTT) which outlive the draining period. These disruptions manifest themselves in multiple ways ranging from explicit disruption (e.g., HTTP 500 error codes) to degraded user experience (e.g., retries). At our scale, we observe that at the tail (i.e., p99.9) most requests are sufficiently large enough to outlive the draining period. In such situations, users need to re-upload or restarts session interactions.

The state of the art for performing updates, i.e., draining and rolling updates, may be suitable for a large number of medium sized online service providers. However, at our scale, the collateral damage of applying these techniques extends beyond our infrastructure and users, to the underlying Internet and cellular fabric.
4.3 Design Choices

With the growing adoption of mono-repos [158] and micro-services [381], the need for a zero-disruption code release mechanism is increasingly becoming important both at Meta and at other companies. In this section, we elaborate on emerging production techniques and discuss how these technique fail to operate at our scale.

4.3.1 Ideal Design (PerfectReload)

Abstractly, there are three ways to seamless update code.

- Language Support (Option-1): The first is to use a language with built-in support for headless updates such as Erlang [40]. Unfortunately, this approach is not supported by most common languages. In fact, only a trivially small number of our services are written in such a language. Orthogonally, the effort required to rewrite our entire fleet of services in such a language is practically unrealistic.

- Kernel Support (Option-2): A more promising alternative is to leverage the support from operating systems for seamlessly migrating connections between two processes – from the one running old code to the one running new code. However, this has two main drawbacks: First, kernel techniques like SO_REUSEPORT do not ensure consistent routing of packets. During such migration of a socket, there is a temporary moment before the old process relinquishes the control over it when both processes will have control over the socket and packets arriving at it. While temporary, such ambiguous moments can have a significant impact at our
scale. To highlight this problem, in Figure 4.5, we examine the number of misrouted packets during a socket handover. These misrouted packets illicit errors code from the server which will propagate to the end-users. While there are solutions to enforce consistency, e.g., CMSG, SCM_RIGHTS, and dup(), they are still limited for UDP. Second, these techniques are not scalable owing to the hundreds of thousands of connections at scale per instance. Additionally, a subset of the socket state, e.g., TLS, may not be copied across process boundaries because of the security implications of transferring sensitive data.

Ignoring the scalability and consistency issues, we note that two key issues remain un-addressed. First, for long lived connections, we may need to wait for an infinite amount of time for these connections to organically drain out, which is impractical. Second, application-specific states are not transferred by SO_REUSEPORT, only new socket instances are added to the same socket family. Terminating these long lived connections or ignoring application state will lead to user facing errors. Thus, we need a general solution for addressing application state and for explicitly migrating long-lived connections.

- Protocol and Kernel Support (Option-3): We can address scalability issues by only migrating listening sockets across application processes and allowing the old instance to drain gracefully. For long-lived connections, we can leverage a third party server such as an upstream component to help drain and coordinate migrations of active connections. For example, during the update of App. Servers, an App. Servers can signal the Proxygen to migrate connections by terminating active connections and setting up new connections. However, for this approach to be disruption free, it requires the connection’s protocol to support graceful shutdown semantics (for e.g. HTTP/2’s GoAways) which not supported by a significant subset of our connections such as HTTP/1.1 or MQTT. Moreover, existing APIs and mechanisms in the kernel are inadequate to achieve such migration of sockets properly without causing disruptions for UDP based applications.

The ideal disruption-free release system is expected to:
1. **Generic**: Generalizes across the plethora of services and protocols. As demonstrated with Options-3 and Options-1, specialized solutions limited to specific languages or protocol do not work at large providers where multiple protocols and languages need to be supported.
2. **Preserve State**: Should either reserve or externalize states to prevent the overhead of rebuilding them after restarts.
3. **Tackle long-lived connections**: Should be able to prevent long-lived connection disruptions as draining period is not enough for them to gracefully conclude.

### 4.4 Zero Downtime Release

In this section, we present the design and implementation details for framework to achieve Zero Downtime Release. Our framework extends on option-3 and introduces primitives to orchestrate scheduling, state externalization, and design enhancements to enable pre-existing operating system to scale to our requirements. In particular, our framework introduces three update mechanisms, and are composed to address the unique requirements of the different tiers in the end-to-end infrastructure. Next, we elaborate on each of these three update/release mechanisms and illustrate how our primitives are used and in particular how the unique
challenges are addressed. The key idea behind the techniques is to leverage Proxygen, either its building blocks (e.g., sockets) or its position in end-to-end hierarchy, to prevent or mask any possible disruption from end-users.

### 4.4.1 Socket Takeover

Socket Takeover enables *Zero Downtime Restarts* for Proxygen by spinning up an updated instance in parallel that takes-over the listening sockets, whereas the old instance goes into graceful draining phase. The new instance assumes the responsibility of serving the new connections and responding to health-check probes from the L4LB Katran. Old connections are served by the older instance until the end of draining period, after which other mechanism (e.g., *Downstream Connection Reuse*) kicks in.

**Implementation:** We work-around the technical limitations discussed in (§ 4.4) by passing file descriptors (FDs) for each listening socket from the older process to the new one so that the listening sockets for each service addresses are never closed (and hence no downtime). As we pass an open FD from the old process to the newly spun one, both the passing and the receiving process share the same file table entry for the listening socket and handle separate accepted connections on which they serve connection level transactions. We leverage the following Linux kernel features to achieve this:

1. **CMSG:** A feature in `sendmsg()` [328, 325] allows sending control messages between local processes (commonly referred to as ancillary data). During the restart of L7LB processes, we use this mechanism to send the set of FDs for all active listening sockets for each VIP (Virtual IP of service) from the active instance to the newly-spun instance. This data is exchanged using `sendmsg(2)` and `recvmsg(2)` [328, 327, 325] over a UNIX domain socket.

2. **SCM_RIGHTS:** We set this option to send open FDs with the data portion containing an integer array of the open FDs. On the receiving side, these FDs behave as though they have been created with `dup(2)` [326].

**Support for UDP protocol:** Transferring a listening socket’s FDs to a new process is a well-known technique [278, 332, 323, 263] has been added to other proxies like (HAProxy [4]) in 2018 [5] and more recently to Envoy [54]. Although the motivation is same, Proxygen’s *Socket Takeover* is more comprehensive as it supports and addresses potential disruptions for multiple transport protocols (TCP and UDP).

Although UDP is connection-less, many applications and protocols built on top of UDP (QUIC, webRTC etc.) do maintain states for each flow. For TCP connections, the separation of listening and accepted sockets by Linux Kernel lessens the burden to maintain consistency from the user-space application. On the other hand, UDP has no such in-built support. Typically, a consistent routing of UDP packets to a socket is achieved via application of hashing function on the source and the destination address for each packet. When `SO_REUSEPORT` socket option is used for an UDP address (VIP), Kernel’s internal representation of the socket ring associated with respective UDP VIP is in flux during a release – new process binds to same address and new entries are added to socket ring, while the old process shutdowns and gets its entries purged from the socket ring. This flux breaks the consistency in picking up a socket for the same 4-tuple combination. This significantly increases the likelihood of UDP packets being misrouted to a different process that does not have state for that flow (Figure 4.5).
Owing to mis-routing of UDP packets, a typical SO_REUSEPORT-less architecture uses a thread dedicated to accepting new connection that hands off newly accepted connections to worker threads. UDP being datagram centric and without any notion of packet type such as SYN for TCP, the kernel cannot discriminate between a packet for a new connection vs an existing one. The approach of using one thread to accept all the packet cannot scale for high loads since the responsibilities of maintaining states and multiplexing of UDP packets must now happen in the application layer, and such thread becomes a bottleneck.

To circumvent the scaling issues, we use SO_REUSEPORT option with multiple server threads accepting and processing the packets independently. To solve the mis-routing issue, we extended the Socket Takeover to pass FDs for all UDP VIP sockets. This essentially is equivalent of calling dup() on an existing socket FD and thus avoids creation of a new FD for this socket. In other words, the socket ring for this VIP within kernel remains unchanged with each process restart. The newly spun process can thus resume processing packets from where the old process left off.

However, one problem still remains un-addressed. All the packets are now routed to the new process, including the ones for connections owned by the old (existing) process. For applications that require and maintain states for each UDP based flow (e.g., QUIC), the new process employs user-space routing and forwards packets to the old process through a pre-configured host local addresses. Decisions for user-space routing of packets are be made based on information present in each UDP packet, such as connection ID that is present in each QUIC packet header. In our implementation this mechanism effectively eliminated all the cases of mis-routing of UDP packets while still being able to leverage multiple threads for better scalability.

**Workflow:** (Figure 4.10) An existing Proxygen process that is serving traffic has already bound and called accept() on socket FDs per VIP. In step A⃝, the old Proxygen instance spawns a Socket Takeover server that is bound to a pre-specified path and the new Proxygen process starts and connects to it. In step B⃝, the Socket Takeover server then sends the list of FDs it has bound, to the new process via the UNIX domain socket with sendmsg() and CMSG. The new Proxygen process listens to the VIP corresponding to the FDs (step C⃝). It then sends confirmation to the old server to start draining the existing connections (step D⃝). Upon receiving the confirmation from the new process, the old process stops handling new connections and starts draining existing connections (step E⃝). In step F⃝, the new instance takes over the responsibility of responding to health-check probes from the L4LB layer (Katran). Step G⃝ stays active until the end of draining period and UDP packets belonging to the older process are routed in user-space to the respective process.

**View from L4 as L7 restarts:** An L4LB (Katran) that sits in front of the L7 proxy and continuously health-checks (HC) each L7LB is agnostic of updates in L7LB servers because of the zero downtime restart mechanism. We can thus isolate the L7 restarts only to that layer. This not only simplifies our regular release process but also help in reliability of the overall system since the blast radius of a buggy release is largely confined to one layer where mitigation (or rollbacks) can be applied swiftly.

**Connections between Edge and Origin:** Proxygen in Edge and Origin maintain long-lived HTTP/2 connections (§ 4.2) between them. Leveraging GOAWAY, they are gracefully terminated over the draining period and the two establish new connections to tunnel user connections and requests without end-user disruption.

**Caveats:** Socket Takeover process is not free-of-cost and incurs CPU and memory overhead, as the two
Proxygen instances run parallel on the same machine. Although the machine stays available to serve new connections, there’s (often insignificant) diminished CPU capacity for the draining duration (§ 4.6.3).

4.4.2 Downstream Connection Reuse

As described earlier, MQTT is used to keep persistent connections with billions of users and the protocol requires underlying transport session to be always available (e.g., for live notifications). As a result, MQTT clients periodically exchange ping and initiate new connections as soon as transport layer sessions are broken. MQTT does not have a built-in disruption avoidance support in case of Proxygen restarts and relies on client re-connects. Given the lack of GOAWAY-like support in the protocol, the edge only has the options of either waiting for the clients to organically close the connection or forcefully close it and rely on client-side reconnects.

A key property for MQTT connections is that the intermediary LBs only relay packets between pub/sub clients and their respective back-ends (pub/sub brokers) and as long as the two are connected, it does not matter which Proxygen relayed the packets. MQTT connections are tunneled through Edge to Origin to MQTT back-ends over HTTP/2 (§ 4.2). Any restarts at Origin are transparent to the end-users as their connections with the Edge remains undisturbed. If we can reconnect Edge Proxygen and MQTT servers through another Origin Proxygen, while the instance in question is restarting, end users do not need to reconnect at all. This property makes Origin Proxygen “stateless” w.r.t MQTT tunnels as the Origin only relays the packets. Leveraging this statelessness, disruptions can be avoided for these protocols that do not natively support it – a mechanism called Downstream Connection Reuse (DCR).

Each end-user has a globally unique ID (user-id) and is used to route the messages from Origin to the right MQTT back-end (having the context of the end-user connection). Consistent hashing [161, 9] is used to keep these mappings consistent at scale and, in case of a restart, another Proxygen instance can take-over the relaying responsibility without any end-user involvement and disruption (at back-end or user side).

Workflow: Figure 4.11 presents the details of the Downstream Connection Reuse transactions. When an Origin Proxygen instance is restarting, it sends a reconnect_solicitation message to downstream Edge LB step A to signal the restart. Instead of dropping the connection, Edge LB sends out re_connect (contains user-id) to Origin, where another healthy LB is used to relay the packets to corresponding back-end (located...
through *user-id* (steps ⃣, ⃤). MQTT back-end looks for the end-user’s connection context and accepts *re_connect* (if one exists) and sends back *connect_ack* (steps ⃣, ⃤). Otherwise, *re_connect* is refused (by sending *connect_refuse*), *Edge* drops the connection and the end-user will re-initiate the connection in the normal way.

**Caveats:** Restarts at *Origin* is the ideal scenario for DCR as *Edge* is the next downstream and can thus keep the restarts transparent to users. For a restart at the *Edge*, the same workflow can be used with end-users, especially mobile clients, to minimize disruptions (by pro-actively re-connecting). Additionally, DCR is possible due to the design choice of tunneling MQTT over HTTP/2, that has in-built graceful shutdown (*GOAWAYs*).

### 4.4.3 Partial Post Replay

An *App. Server* restart result in disruption for HTTP requests. Due to their brief draining periods (10-15s), long uploads (POST requests) pose a specific pain point and are responsible for the most disruptions during restarts.

A user’s POST request makes its way to the *App. Server* through the *Edge* and *Origin Proxygen*. When the app. server restarts, it can react in multiple ways:

1. The *App. Server* fails the request, responds with a 500 code and the HTTP error code makes it way to user. The end-user observes the disruption in form of “Internal Server Error” and the request gets terminated (disrupted).
2. The *App. Server* can fail the request with 307 code (Temporary Redirect), leading to a request re-try from scratch over high-RTT WAN (performance overhead).
3. The 307 redirect can be improved by buffering the POST request data at the *Origin L7LB*. Instead of propagating the error (500 or 307 retry) to the user, the *Origin L7LB* can retry the buffered request to another healthy *App. Server*. The massive overhead of buffering every POST request until completion makes this option impractical.

In light of deficiencies of alternate effective techniques, *Partial Post Replay* [195] leverages the existence of a downstream *Proxygen* and the request data at restarting app. server to *hand-over* the incomplete, in-process requests to another *App. Server*. A new HTTP code (379) is introduced, used by *App. Server* to indicate a restart to the downstream *Proxygen*.

**Workflow:** Figure 4.12 presents a high-level view of the workflow. A user makes a POST request (⃣) which is forwarded to an *App. Server* (⃤). When the *App. Server* (AS in Figure 4.12) restarts, it responds
to any unprocessed requests with incomplete bodies by sending a 379 status code and the partially received
POST request data, back to downstream Origin Proxygen (©). For HTTP/1.1, the status message is set to
“Partial POST Replay”. The Proxygen instance does not send the 379 status code to the end-user and instead
builds the original request and replays it to another healthy App. Server (©). When the request gets completed,
the success code is returned back to the end-user (©).

**Caveats:** There is a bandwidth overhead associated with replaying the request data — high bandwidth
connections, existing between Origin Proxygen and App. Server in DataCenter, are required to make Partial
Post Replay viable. Additionally, since end-user applications are not expected to understand code 379, it
should never be sent back to end-user. In case when intermediary cannot replay request to another server, the
requests should be failed with standard 500 code.

### 4.4.4 Applicability Considerations

Next, we discuss the applicability of the three mechanisms to the different tiers. As mentioned in § 4.2, the
tiers differ in their functionality and operational capacity, and can also have specific operational limitations,
e.g., App. Server tier is restarted hundreds of times a week and requires draining period in order of tens of
seconds. Such operational aspects decide the applicability of the three mechanism.

Whereas Socket Takeover is used at every Proxygen instance in the end-to-end architecture, Socket Takeover
is not used for the HHVM server at the App. Server tier and Partial Post Replay is used there to tackle any
disruptions. Due to the very brief draining period for HHVM servers, Socket Takeover by itself is inadequate
to prevent disruptions for large POST requests as these requests are not expected to be completed by the
end of the draining phase for the old instance and hence would lead to connection resets when the old in-
stance terminates. Therefore, a co-ordinated form of draining with the downstream is required to hand-over
the request to another healthy instance. Additionally, operational aspects of App. Server tier makes Socket
Takeover unfavorable as these machines are too constrained along CPU and memory dimensions to support
two parallel instances (e.g., priming local cache for a new HHVM instance is memory-heavy). Downstream
Connection Reuse is used at both Edge and Origin Proxygen for MQTT-based services due to their long-lived
connection nature. For Downstream Connection Reuse to work at Edge, application code at the user-end
needs to understand the connection-reuse workflow transactions.

The three mechanisms differ with respect to the protocol or the target layer in the networking stack. Hence, there’s no interdependencies and the mechanisms are used concurrently. Socket Takeover and Down-
stream Connection Reuse are triggered at the same time, i.e., when a Proxygen restarts. Note that, if the
next-selected machine to relay the MQTT connections is also under-going a restart, it does not have any
impact on Downstream Connection Reuse, since the updated, parallel instance is responsible for handling
all the new connections. For Partial Post Replay, it is possible that the next HHVM server is also restarting
and cannot handle the partially-posted requested, since the corresponding instance is in draining mode and
not accepting any new requests. In such a case, the downstream Proxygen retries the request with a different
HHVM server. At production, the number of retries is set to 10 and is found enough to never result in a failure
due to unavailability of active HHVM server.
4.5 Hands-on Experience

In this section we discuss our experiences and the challenges we faced in developing and deploying the Zero Downtime Release.

4.5.1 Socket Takeover

As discussed in section 4.4.1, passing controls of a listening socket between processes with SCM_RIGHTS is known technique. Our solutions use it as a building block for zero-downtime release in a large-scale production service. It is thus important to understand limitations and operational downsides of using such feature and have proper monitoring and mitigation steps in place.

- **Downsides of sharing existing sockets:** When the ownership of a socket is transferred to another process its associated state within the kernel remains unchanged since the File Descriptor (FD) still points to the same underlying data structure for the socket within Kernel even though the application states in user-space have changed with the new process taking over the ownership of the socket.

  While this mechanism to pass control of a socket is not an issue *per se*, an unchanged socket state in the Kernel even after restart of the associated application process is not only unintuitive but can also hinder in debugging of potential issues and prevent their swift mitigations. It is a common practice to roll back the newly released software to a last known version to mitigate ongoing issues. Albeit rare in occurrence, if there is any issue in the Kernel involving socket states, such as ones stemming from erroneous states within their data structure, diagnosing or mitigating such issues on a large scale fleet poses several challenges since a rollback of the latest deployment does not resolve the issue.

  For example, we encountered a bug in the UDP write path [124] after enabling the generic segmentation offload (UDP GSO) Kernel feature [125]. On many of the updated servers, the buffer in UDP sockets (*sk_buff*) were not cleared properly after failures within certain *syscalls* in write paths. This resulted in slow accumulation of buffers over period of time, eventually leading to a system wide failure to successfully write any further packet. A simple restart of the application process owning the socket did not mitigate this issue since the underlying socket structure persisted across process restarts while using the Socket Takeover mechanism.

  Another pitfall with the mechanism of passing the control of sockets is that it introduces possibilities of leaking sockets and their associated resources. Passing the ownership of sockets to a different process using their *FDs* is essentially equivalent to a system call *dup(fd)* wherein upon passing these *FDs* to the new process, the Kernel internally increases their reference counts and keeps the underlying sockets alive even after the termination of the application process that owns them. It is thus essential that the receiving process acts upon each of the received *FDs*, either by listening on those sockets or by closing any unused ones.

  For example, when multiple sockets bind to the same address using the socket option *SO_REUSEPORT*, the Linux Kernel internally multiplexes incoming packets to each of these sockets in a fairly uniform manner. However, during a release if the newly spun process erroneously ignores any of the received *FDs* for these sockets after Socket Takeover, without either closing the *FDs* or listening on the received sockets for incoming
packets, it can lead to large increase in user facing errors such as connection timeouts. This is because the orphaned sockets now owned by the new process are still kept alive in the Kernel layer and hence receive their share of incoming packets and new connections - which only sit idle on their queues and never get processed.

**Remediation:** For swift mitigations of operational issues during a release, such as the ones involving persistent erroneous socket states or socket leaks as mentioned earlier in this section, a mechanism to dynamically disable the *Socket Takeover* process presents a safe and convenient approach. To allow such options, applications employing *Socket Takeover* mechanism must incorporate such fall-back alternatives on all of its phases, spanning from the initialization phase (when sockets are created) to the shutdown phase (as the process gracefully terminates all of its existing connections). *Proxygen*, for example, allows transitioning to either modes by executing a set of heuristics tuned to its needs on each of its operating phases. During the initialization phase *Proxygen* resorts to allocating and binding new sockets if it cannot receive *FDs* of listening sockets from the existing process. Similarly, during its shutdown phase *Proxygen* adopts different strategies depending on the runtime behavior - such as signaling the *L4LB* to not send any new traffic by failing the health probes if no other instance is running on the same host, to transitioning back to the running mode from the shutdown phase if the newly spun process crashes during its initialization phase. Additionally, being able to selectively disable this approach for a subset of sockets makes ad-hoc debugging and development easier.

We recommend fine-grained monitoring of socket states, such as connection queue lengths, failure counts in socket reads and writes, number of packets processed and dropped and counts of *FDs* passed across process domains to detect problems of these nature in early phases. While the occurrence of such issues is quite rare in our experience, monitoring of such events not only notifies the concerned engineers but can also trigger automated responses to mitigate the impact.

- **Premature termination of existing process:** During a release phase, there is a period of time during which multiple application processes within a same host are serving requests. During such window of time, premature exit of the old instance before the new service instance is completely ready and healthy can result in a severe outage. Typically, the newly spun instance goes through initialization and setup phase, such as loading of configuration, setting up of its server pools and spinning of threads, and may also invoke remote service calls. It is thus common to encounter issues due to issues of its own such as deadlocks and memory leaks or due to issues external to itself, such as failures on the remote service calls in its startup path. It is thus imperative to keep the old service instance and serve incoming requests as the new instance is being initialized.

**Remediation:** Our recommendation based on experience is to implement a message exchange mechanism between the new and old application server instances to explicitly confirm that the *Socket Takeover* was successful, and acknowledge the readiness to serve. The same mechanism that was used to pass *FDs* of sockets during the *Socket Takeover* can be trivially extended to allow exchange of such messages. Only upon receiving the explicit message should the old instance initiate its shutdown phase with proper draining phase to allow graceful termination of existing connections.

- **Backward compatibility:** Any changes in the *Socket Takeover* mechanism must be backward compatible and support versioning. If the system of acknowledgment from the new instance to the old instance
breaks, the service gets exposed to significant availability risk. To illustrate this notion further, any change in the behavior of the *Socket Takeover* version $n$ needs to be compatible with the version $n + 1$, and thus, needs to handle all the possible transitions:

- $n \rightarrow n$ restart of an existing version in production
- $n \rightarrow n + 1$ deployment of the new implementation
- $n + 1 \rightarrow n$ revert of the deployment above (for example, due to a regression)
- $n + 1 \rightarrow n + 1$ restart of the new version in production

For this reason we recommend the implementation of explicit versioning logic within the messaging mechanism to make it easier for the developer to specify the intended behavior when introducing new changes in the algorithm. A newly spun process can gracefully terminate itself early if it fails to negotiate a common version with an existing process. Only after successfully completing version negotiation, the existing process sends the *FDs* of listening sockets to the new process.

- **Availability risks and health-checks:** The process of release itself can increase the availability risks for a service going through the release. For instance, degradation in the health of a service being released even at a micro level, such as one at an individual host level, can escalate to a system wide availability risks while updating a large fraction of its servers. The overall release process, therefore, must be tuned such that the health of the service being updated remains consistent for an external observer of the service. For example, when *Proxygen*, an *L7LB* instance, is being updated with the *Socket Takeover* mechanism, its state must continue to be perceived as *healthy* to the observers in *L4LB* layer. Occasionally it is possible that the servers going through deployment in peak hours suffer momentary CPU and memory pressure, and consequently reply back as unhealthy to external health monitors for the service. This seemingly momentary flap can escalate to system wide instability due to mis-routing of packets for existing connections if, for example, the *L4LB* layer employs a consistent routing mechanism such as consistent-hash to pick an *L7LB* destination server based on the source and destination addresses in a packet header.

**Remediation:** To avoid instability in routing due to momentary shuffle in the routing topology, such as changes in the list of healthy servers going through a release process using the *Socket Takeover* mechanism,
we recommend adopting a connection table cache for the most recent flows. In Meta we employ a Least Recently Used (LRU) cache in the Katran (L4LB layer) to absorb such momentary shuffles and facilitate connections to be routed consistently to the same end server. Adoption of such mechanism also usually yields performance improvements.

4.5.2 Partial Post Replay

Partial Post Replay is the mechanism we use to hide an app. server restart from a client performing a long upload. Here we discuss potential pitfalls of the store-and-replay solution regarding HTTP semantics and app. server behavior; while some of the solutions were part of the original design, some others are less obvious and were only discovered after deploying the solution in production and noticing interesting errors or outliers.

- **Preserving HTTP Semantics**: Partial Post Replay is designed to work with any HTTP version; some simple rules must be defined for each protocol version to make sure that the necessary state is transferred back to the proxy so that the original request can be replayed to a different server. As an example HTTP/2 and HTTP/3 request pseudo-headers (beginning with ':) are echoed in the response message with a special prefix (e.g. 'pseudo-echo-path:' for the 'path:' pseudo-header). The most interesting corner cases however were discovered with HTTP/1.1 and chunked transfer encoding where the body stream is split into chunks; each chunk is preceded by a header that indicates the chunk length and the end of the body is signaled by a chunk trailer. A proxy implementing PPR must remember the exact state of forwarding the body to the original server, whether it is in the middle or at the beginning of a chunk in order to reconstitute the original chunk headers or recompute them from the current state.

- **Trust the app. server, but always double-check**: A solution like PPR requires a Proxy and its immediate upstream hop to collaborate and implement the client and server-side of the mechanism. In Facebook infrastructure since we control both sides there is implicit trust on the app. server doing the right thing and not be malicious. However, the upstream may also behave as a proxy itself forwarding responses from another app. server which does not implement PPR and may be using the HTTP response status code 379. We hit this
case in production, where the web-server acting as a proxy would return responses from a buggy upstream service returning randomized HTTP response codes due to a memory corruption bug. Although this was the result of a bug, we realized that there was the need for a more strict check on the conditions to enable the feature on a specific request.

**Remediation:** The current implementation and RFC do not define a negotiation mechanism for the feature and assumes previous knowledge at the intermediary and server that the peer is going to support the feature. Also, HTTP response code 379 was specifically picked within an unreserved range in the IANA status code registry [8] and therefore no assumption can be made on the server not using that status code for other purposes. To disambiguate then we used the HTTP Status message, and defined that the proxy must enable PPR only on seeing a 379 response code with `PartialPOST` as the status message.

### 4.6 Evaluation

Our evaluation of our framework, **Zero Downtime Release**, is motivated by the following practical questions:

1. How does **Zero Downtime Release** fare in comparison with traditional release techniques on performance
and availability grounds?

2. What are the operational benefits of using Zero Downtime Release at productions scale in terms of minimizing disruptions, preserving capacity and release scheduling?

3. What are the system overheads of Zero Downtime Release?

**Evaluation Metrics**

Zero Downtime Release has been operational at Meta for multiple years and has assisted in rolling-out thousands of code updates with minimal disruptions. A sophisticated auditing infrastructure [395, 298] has been built over the years for real-time monitoring of cluster and user performance, including releases and their implications. Each restarting instance emits a signal through which its status can be observed in real-time (e.g., health of the parallel processes, duration of takeover etc.). The instances also log system benchmarks (e.g., CPU utilization, throughput, Request per Second (RPS) served etc.) as well as counters for the different connections (e.g., Number of MQTT connections, HTTP status code sent, TCP RSTs sent etc.). The monitoring systems also collect performance metrics from the end-user applications and serve as the source of measuring client-side disruptions (e.g., errors, HTTP codes sent to user etc.).

Our evaluation of Zero Downtime Release, we examine these data sources to analyze the performance and operational implications of our Zero Downtime Release framework.

**Evaluation Setup**

In our evaluation, we conduct experiments in production clusters across the globe, serving live end-user traffic. Experimenting with live deployments allows us to not only measure the impact at scale, but also measure the impacts across the different protocols. For each system component, we aim to highlight improvement in target system’s availability, quality of service (QoS) and their impact on client. We further explore their use in alleviating the complexity of managing hundreds of production clusters. Finally, we address the additional costs related to persistent long haul techniques and explore their impact on performance.

### 4.6.1 Comparison with Traditional Release

To measure the effectiveness of Zero Downtime Release, we conducted multiple HardRestart across productions clusters. A HardRestart mirrors the traditional roll-out process — updates are rolled out in batches across a cluster and the restarting instances enter the draining mode (i.e., the server stops receiving new connection until the end of draining period). Since the goal is to compare against Zero Downtime Release, we set the same draining duration (20 minutes) and test two batch sizes (5% and 20%) in ten randomly selected Edge production clusters. During both restart strategies, we monitor both system metrics (e.g., idle CPU) and performance counters (e.g., HTTP and MQTT connection stats). Furthermore, we analyzed the client-side disruptions by examining performance metrics collected from end-users.

**Improved time to completion**

Figure 4.21 summarizes Completion Times of various restart mechanisms for Proxygen and App. Server releases (i.e., time required to update our global deployments for either Proxygen and App. Server). We observe that in the median update, Proxygen releases finish in 1.5 hours, whereas, App. Server releases are

---

1 comparison with traditional (§ 4.6.1) covers restarts, repeated across 10 cluster, while for operational benefits we analyze historical data across 66 cluster restarts (§ 4.6.2) across the globe.
even faster (25 minutes). The major factor behind the differences in their completion time is the different draining behavior. Proxygen are configured to drain for 20 minutes while App. Server have a short draining interval (10-15 seconds) since their workload is dominated by short-lived requests. As we are going to show next, Zero Downtime Release preserves capacity and minimizes while taking order of tens of minutes to restart the tiers.

**Improved L7 Cluster Capacity**

Katran maintains an updated view of available Proxygen through health-checks. Recall that performing a HardRestart on an instance causes this instance to block new connections and thus to fail health-checks, because health-check connections are rejected. Whereas Zero Downtime Release enables the new Proxygen instance to take-over health-check responsibility. Looking at Katran logs, we observe the expected behavior: Zero Downtime Restart stays transparent to Katran while, for HardRestart, the restarted instances are removed from Katran table.

To explore the impact of the two restart approaches on clusters’ available capacity, we measure the idle CPU metrics under the draining phase of both restart approaches. Figure 4.13(b) plots the cluster’s total idle CPU resources, normalized by the baseline idle CPU resources, recorded right before the restart. In Socket Takeover (§ 4.6.3), we expect an increase in CPU usage because of the parallel process on same machine, leading to a slight (within 1%) decrease in cluster’s idle CPU. However, this is radically different from the HardRestart case, where the cluster’s CPU power degrades linearly with the proportion of instances restarted because each instance is completely taken offline.

**Minimizing User Faced Disruptions**

**Pub/Sub services (Downstream Connection Reuse):**

To measure MQTT related disruptions, we performed restarts with and without Downstream Connection Reuse at the Origin. Figure 4.14 highlights its impact on minimizing the client side disruptions. The figure plots a timeline of Publish messages routed through the tunnel to measure the impact of restarts on communication between end-users and their MQTT brokers (back-ends). The figure also plots the median number of new MQTT connections created at the back-ends, by measuring the number of ACKs sent in response to MQTT connect messages from end-users. The number represent the median across the cluster machines and are normalized by their value right before restart. In contrast to DCR case where the number of published messages do not deteriorate during the restart, we observer a sharp drop in Publish messages when Downstream Connection Reuse is not used (woutDCR), indicating disruptions in communication between users and their MQTT brokers. On the other hand, we observe a sharp spike in number of ACKs sent for new MQTT connections for woutDCR case, indicating that the restarting instance terminated the MQTT connection, leading to the clients retrying to reconnect with the back-end brokers. With DCR, we do not observe any change as connections between users and their back-end brokers are not terminated and, instead, are routed through another Origin Proxygen to same broker.

**Web (Partial Post Replay):** In absence of Partial Post Replay, the restarting App. Server terminates the in-process POST requests and returns error code (e.g., HTTP code 500) to the downstream Proxygen and, eventually, the disruption (error code) reaches the end-user. In case of Partial Post Replay, the server returns
a code 379 and the partial POST data which is then replayed to another App. Server alongside the original request.

To test Partial Post Replay’s effectiveness, we observe App. Server restarts from the downstream Origin Proxygen’s vantage point and inspect the POST requests sent to a restarting server. A reception of 379 response, along with the partial request data, signals a request that would have faced disruption in the absence of Partial Post Replay— allowing us to measure the scale of disruptions due to App. Server restarts. Figure 4.16 compares the Partial Post Replay’s impact by presenting the percentage of connections disrupted across the web tier for 7 days. Note that App. Server are restarted tens of times a day (Figure 4.2) and the 7 days worth of data covers around 70 web tier restarts. We observe that Partial Post Replay is extremely effective at minimizing the POST requests disruptions. Although the percentage might seem very small (e.g., 0.0008 at median), there are billions of POST request per minute for the entire web-tier and even the small percentages translate to huge number of requests (e.g., ~6.8 million for median).

Minimizing Proxy Errors

A major benefit of using Zero Downtime Release is to improve proxy performance during restart w.r.t. errors. Errors result in connection terminations or 500 response codes both of which are highly disruptive to end-user’s performance and QoE. To measure these disruptions, we measured the errors sent by the Edge proxy to end-users, under both kind of restarts. Figure 4.17 presents the ratio of errors observed for the two restarts (traditional and Zero Downtime Release). The 4 types of errors correspond to different types of disruptions: (i) Connection Reset (conn. rst.) refers to sending a TCP RST to terminate the connection, (ii) Stream abort/unacknowledged refers to errors in HTTP, (iii) Timeouts refer to TCP level timeouts, (iv) write timeout refers to case when application times-out due to disruption in underlying connection. We observe a significant increase in all errors for “traditional” as compared to Zero Downtime Release. Write timeouts increase by as much as 16X and are significantly disruptive for user experience as users can not retry right away.
Impact on consistent packet routing

Next, we measure the efficacy of *Socket Takeover* for consistently routing packets to the right proxy process, in cases where multiple proxies are available (updated and draining instance). We disable the connection-ID based QUIC packet routing and inspect the packets received at both instances. Since the HardRestart case has only one instance running at a time, no UDP mis-routing exists. In the context of this experiment, *traditional* approach refers to the case where sockets are migrated to the updated instance, but the system lacks connection-ID powered user-space routing. Figure 4.15 present the number of UDP packets mis-routed per instance. A packet is marked mis-routed if the wrong proxy instance receives it i.e packets bound for the draining instance are received at updated one. Although we observe some mis-routing for *Zero Downtime Release* at start of the restart, their magnitude is insignificant compared to traditional case, with 100X less packets mis-routed for the worst case (tail at T=2).

4.6.2 Operational Benefits at Scale

To evaluate the effectiveness of *Zero Downtime Release* at scale, we monitored 66 production cluster restarts across the globe. *Proxygen* releases are rolled-out in batches of 20% instances restarted at a time, with a 20 minute draining period.
Performance and stability improvements:

Figure 4.18 shows a timeline of the system and performance metrics (Requests Per Second (RPS), number of active MQTT conn., throughput and CPU utilization) during releases. The metrics are normalized by the value just before the release was rolled-out. During each batch restart, we collected the target metrics from each cluster instance and Figure 4.18 plots the distributions (averaged over a minute) observed across two groups of machines: (i) the 20% restarted ($G_R$), (ii) the rest of 80% non-restarted ($G_{NR}$). Observing the two groups side by side demonstrates standing their behavior during restarts. The timeline (x-axis in minutes) marks 4 phases: (i) $T \leq 1$ state before restart, (ii) $T=2$ marks the start of batch restart, (iii) $T=24$ marks the end of draining period, (iv) $T \geq 24$ state after batch restart is concluded. All the observed metrics are normalized by their values at $T=0$. We further present a cluster-wide view in form of the average metrics calculated across all instances of the cluster.

Cluster-wide behavior: Across RPS and number of MQTT conn., we observe virtually no change in cluster-wide average over the restart period. No significant change in these cluster-wide metrics after $T=2$, even with 20% of the cluster restarting, this highlights the benefits of Zero Downtime Release at scale in practice. We do observe a small increase in CPU utilization after $T=2$, attributed to the system overheads of Socket Takeover, i.e., two Proxygen instances run parallel for the duration of draining period on same machine resources ($\S$ 4.6.3).

$G_R$ vs $G_{NR}$ behavior: Analyzing the per-group breakdown, we observe the inflation in CPU utilization for restarting instances ($G_R$) only persists for two minutes (at $T=2,3$) and the CPU util. gradually decreases until the end of draining period where we observe a sharp decrease (at $T \geq 24$) due to termination of parallel process. CPU util. of the ($G_R$) to be lower than cluster-wide average and $G_R$ (non-restarted group) is surprising as every machine in $G_R$ runs two Proxygen instances during $2 \leq T \leq 24$. We observe RPS to drop for $G_R$ and rise for $G_{NR}$ after $T=3$, indicating that $G_R$ instances are serving lower number of requests than their pre-restart state and the $G_{NR}$ instances are serving a higher proportion. The contrasting behavior for the two groups arise due to CPU being a function of RPS i-e an instance serving less number of requests requires lower CPU cycles. Since Katran uniformly load-balances across Proxygen fleet and the newly-spun, updated instance has no request backlog, it gets the same share of requests as others — leading to the drop and ramp-up in RPS over time.
For MQTT connections, we observe their number to fall across $G_R$ instances and gradually rise for $G_{NR}$. This behavior is expected as the MQTT connections get migrated to other healthy instances ($G_{NR}$) through *Downstream Connection Reuse*. However, we do not observe their number to drop to zero for $G_R$ at end of draining as the updated, parallel *Proxygen* picks up new MQTT connections during this duration.

**Timeline for disruption metrics**: Figure 4.19 builds a similar timeline for disruption metrics – TCP resets (RST), HTTP errors (500 codes) and Proxy errors, presenting their count. Each point is the average value observed for a cluster and the box plot plots the distribution across the clusters. The timeline is divided into the four phases, similar to Figure 4.18. We observe that the disruption metrics stay consistent throughout the restart duration. Even for a 20% restart, we do not observe any increase in these disruption metrics — highlighting the effectiveness of *Zero Downtime Release* in shielding disruptions. No change in TCP RSTs highlights the efficacy of *Zero Downtime Release* for preventing TCP SYN related inconsistencies, observed for *SO_REUSEPORT* based socket takeover techniques [323].

**Ability to release at peak-hours**

Traffic load at a cluster changes throughout the day (exhibiting di-urinal pattern [355]). The traditional way is to release updates during off-peak hours (e.g., midnight) so that the load and possible disruptions are low. Figure 4.20 plots the PDF of *Proxygen* and *App. Server* restarts over the 24 hours of the day. *Proxygen* updates are mostly released during peak-hours (12pm-5pm). Whereas, the higher frequency of updates for *App. Server* (Figure 4.2) leads to a continuous cycle of updates for the *App. Server* — a fraction of *App. Server* are always restarting throughout the day as seen by the flat PDF in Figure 4.20. From an operational perspective, operators are expected to be hands-on during the peak-hours and the ability to release during these hours go a long way as developers can swiftly investigate and solve any problems due to a faulty release.

**4.6.3 System Overheads**

**Micro-benchmarks**: Improving cluster availability and client’s performance during a restart can come at the cost of increased system resource usage. Figure 4.22 plots the system resource usage during the restart phase for machines in a randomly chosen production edge cluster. Since the CPU consumption is variable at different phases of the restart (increases at first and then returns to normal state as seen in timeline figure 4.18), we plot system benchmarks during the entire restart and present the median numbers observed across the different machines in a randomly selected edge cluster. The presence of two concurrent *Proxygen* instances contributes to the costs in system resources (increased CPU and Memory usage, decreased throughput). The change in throughput correlates with CPU usage (inverse proportionally), and the tail throughput decreases is caused by the initial spike in CPU usage. Although the tail resource usage can be high (persisting for around 60-70 seconds), the median is below 5% for CPU and RAM usage i-e the increased resource usage does not persistent for the whole draining duration (§ 4.6.2). As the machine is still available and able to serve connections, this overhead is a small price to pay for minimizing disruptions and keeping the overall cluster close to its baseline capacity (i.e., non-restart scenario).
4.7 Conclusion

Owing to high code volatility, CSPs release up to tens of updates daily to their millions of globally-distributed servers and the frequent restarts can degrade cluster capacity and are disruptive to user experience. Leveraging the end-to-end control over a CSP’s infrastructure, the paper introduces Zero Downtime Release, a framework to enable capacity preservation and disruption-free releases, by signaling and orchestrating connection hand-over during restart (to a parallel process or upstream component). The framework enhances pre-existing kernel-based mechanisms to fit diverse protocols and introduces novel enhancements on implementation and protocol fronts to allow fast, zero-downtime update cycles (globally-distributed fleet restarted in 25 minutes), while shielding millions of end-users from disruptions.
Chapter 5

Configanator: A Data-driven Approach to Improving CDN Performance.

5.1 Introduction

Web page performance significantly impacts the revenue of content distribution networks (CDNs) (e.g., Facebook, Akamai, or Google), with studies showing that a 100ms decrease in page load times (PLT) can lead to 8% better conversion rate for retail sites [145, 78]. Yet, uniformly improving web performance is becoming increasingly challenging due to the growing disparity in the network conditions (e.g., bandwidth, RTT) [464, 520, 27, 165] and end-user devices [366, 429, 519, 371]. To address this disparity and improve the quality of experience (QoE), the networking community is constantly developing new protocols and configuration parameters for web servers (AKA, edge servers), e.g., PCC [149], BBR [95], QUIC [212], etc.

The optimal choice of configurations is contingent on the network infrastructure [520, 165, 464, 27, 383, 543, 305], website complexity [85, 88, 375, 521, 519], and end-user devices [20, 429, 371]. Furthermore, innovations along any one of these dimensions will lead to changes to default parameters and new protocols. Although different regions and ISPs have radically different networking infrastructure and mobile devices [20, 366], a majority of CDNs continue to employ a “one-size-fits-all” [208] approach to configuring their edge servers, which results in sub-optimal performance [520, 165, 27] and high tail-latency in certain regions [543].

5.1.1 Configuration Tuning Status-Quo

Most attempts to tackle this growing diversity involve manually analyzing the performance of configuration options across different regions [208], devices [20], or websites [520, 432]. While several CDNs expose configuration knobs to their customers [180, 193], it is challenging to take the full advantage of the knobs due to the required manual efforts and the lack of automated learning techniques for effective tuning.

This paper focuses on tuning a broad set of configuration knobs across the transport (e.g., congestion control algorithm) and application layers (e.g., HTTP version) as highlighted in Table 5.1. Next, we illustrate
the challenges and benefits of dynamically tuning network configurations.

**Challenges in tuning stack:** In Figure 5.1, we illustrate the difficulty of tuning configurations by comparing page load time (PLT) of popular websites, when configured using popular tuning techniques. Specifically, Bayesian Optimization [404] (e.g., CherryPick [29]) a statistical technique used for tuning systems configurations [29, 300, 151, 504], operator hand-tuned configurations (discussed in § 5.2), TCP connection reuse a traditional optimization (discussed in § 4.4.2), and a closed-loop offline-learning technique. We compare their performance against two baselines: optimal configuration discovered through an exhaustive brute-force search, and default configurations for Linux and Apache (Table 5.1).

Hand-tuned configurations are manually selected and are thus, coarse-grained. While they out-perform the default at median, they fail to provide optimal performance across varying network conditions and may even lead to performance degradation for some networks. TCP connection reuse only optimizes a subset of knobs (e.g., initial congestion window) and is unable to take full advantage of the diverse network stack knobs. Bayesian optimization aims to quickly discover “good” configuration. While fine-grained, this approach is relatively static and does not re-evaluate old choices, and is thus unable to adapt to network dynamics [310]. We observe the effects of this rigid behavior with wildly varying tail performance. Lastly, we explore an offline model which learns on traces from prior days and applies the learned model on connections for the next day. Offline modeling is fine-grained but with limited dynamicty: the trained model is unable to react to real-time issues. Unfortunately, due to the high dimensionality of the Internet’s dynamics, these real-time issues are the norm, not the exception [258, 255, 310]. We observe in Figure 5.1 that offline performs closest to the optimal but still falls short because of its inability to react in real-time.

Our brief analysis of tuning approaches highlights the need for a dynamic, fine-grained approach to tuning configurations.

### 5.1.2 Configanator

In this paper, we eschew the notion of a homogeneous approach to tuning web server configurations and instead argue for a “curated” approach for configuring on a per-connection basis. In particular, we argue that edge servers should be configured to serve each of the incoming connections with the optimal protocols and configuration parameters, e.g., a web server may employ Cubic in favor of BBR when serving a low
Table 5.1: Web stack configuration parameters.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Option</th>
<th>Default</th>
<th>Example parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport</td>
<td>congestion_control (CC)</td>
<td>Cubic</td>
<td>BBR, Cubic, Reno</td>
</tr>
<tr>
<td></td>
<td>initial congestion window</td>
<td>10 MSS</td>
<td>Integer (1-4, 30)</td>
</tr>
<tr>
<td></td>
<td>slow_start_after_idle</td>
<td>1</td>
<td>boolean {true, false}</td>
</tr>
<tr>
<td></td>
<td>low_latency</td>
<td>0</td>
<td>boolean {true, false}</td>
</tr>
<tr>
<td></td>
<td>autocorking</td>
<td>1</td>
<td>boolean {true, false}</td>
</tr>
<tr>
<td></td>
<td>initRTO</td>
<td>1</td>
<td>decimal (0, 1, 10^4)</td>
</tr>
<tr>
<td></td>
<td>pacing (fair-queue)</td>
<td>0</td>
<td>boolean {true, false}</td>
</tr>
<tr>
<td></td>
<td>timestamps</td>
<td>1</td>
<td>boolean {true, false}</td>
</tr>
<tr>
<td></td>
<td>wmem</td>
<td>(4096)B</td>
<td>[163840]B</td>
</tr>
<tr>
<td>Web App</td>
<td>HTTP Protocol</td>
<td>1.1</td>
<td>1.1, 2</td>
</tr>
<tr>
<td></td>
<td>H2 push</td>
<td>On</td>
<td>On, Off</td>
</tr>
<tr>
<td></td>
<td>H2 max header list size</td>
<td>16384B</td>
<td>Integer values</td>
</tr>
<tr>
<td></td>
<td>H2 header table size</td>
<td>4096B</td>
<td>Integer values</td>
</tr>
<tr>
<td></td>
<td>H2 max concurrent streams</td>
<td>100</td>
<td>Integer values</td>
</tr>
<tr>
<td></td>
<td>H2 initial window size</td>
<td>65535B</td>
<td>Integer values</td>
</tr>
<tr>
<td></td>
<td>H2 max frame size</td>
<td>16384B</td>
<td>Integer values</td>
</tr>
</tbody>
</table>

bottleneck buffer connection [445, 433]. To this end, we argue for a simple but robust server architecture that introduces flexibility into the network stack, enables reconfiguration, and systematically controls configuration heterogeneity. We also introduce a contextual multi-armed bandit based learning algorithm, an embodiment of domain-specific insights, which tunes configuration in a principled manner to find optimal configurations in minimal time. Taken together the design and the learning algorithm, our system, Configanator, enables a CDN to systematically explore heterogeneity in a dynamic and fine-grained manner while improving end-user performance. The design of Configanator faces several practical challenges:

- **Network dynamics**: network may change every few minutes [554, 321, 255] and thus requires continuous learning.
- **Non-Gaussian noise**: CDNs focus on improving tail latency [553, 222, 119] which is often caused by non-Gaussian processes (e.g., last-mile contention [477], mobile device limitations) and are difficult to model.
- **High-Dimensionality**: Content personalization, diverse devices [371, 519], and last-mile connections [477] introduce high dimensionality that limits the efficacy of offline closed-loop approaches [255, 256].
- **High data cost**: Generating data for learning requires testing configurations and may disrupt user’s performance. Hence, the negative impact on users must be minimized.
- **Limited flexibility**: Linux kernel and modern web servers lack the flexibility to tune configurations on a per-connection basis, thus requires enhancing the traditional networking stack.

Table 5.2: Heterogeneity in configs. across 5 regions

The key insight of Configanator is to simultaneously operate in two modes depending on the “quality” of the performance model. Essentially, Configanator intelligently selects samples that speed up model convergence, then at steady-state it transitions into a greedy-mode that stochastically samples points to iteratively
improve performance. *Configanator* further clusters similar connections together and samples across clusters to amortize the cost of exploration.

*Configanator* uses a contextual multi-armed bandit [513] designed explicitly to continuously converge to an optimal (or near-optimal) configuration within a minimal number of exploration steps. Our ensemble fuses the stateful exploration of Gaussian-bandit with the non-determinism of Epsilon-bandit, enabling informed exploration of the configuration space while randomly re-sampling old configurations. The re-evaluation of data samples enables *Configanator* to directly tackle non-Gaussian noise within the domain. The data collected by the ensemble is encoded in a decision tree – which enables quick and easy classification but is also amenable to automatic generation of rules for a CDN’s web server.

To demonstrate the benefits, we conducted large-scale simulations and live deployments. We used datasets from a *GlobalCDN* and public datasets from CAIDA [89], MAWI [45], Pantheon [543] and FCC [181]. Our simulation results show that *Configanator* provides 32-67% (up to 1500ms) improvement in the PLT at tail (p95) across the different traces. Given the recent arms race by CDNs to improve web performance, we believe that *Configanator*’s modest improvements will result in significant revenue savings [78, 145, 396, 82]. Please refer to the project website\(^1\) for the related resources.

### 5.2 Empirical Study

Next, we analyze CDNs to determine the current extent of configuration tuning (§ 5.2.1) and quantify its implications (§ 5.2.2).

#### 5.2.1 Fingerprinting web configurations

We aim to understand if modern CDNs employ homogeneous configurations, as suggested by anecdotal evidence, or heterogeneous configurations to tackle diversity in the Internet’s ecosystem. To this end, we developed a tool to infer and fingerprint a web server’s [208, 364] application/L7 and transport/L4 layers configuration parameters by actively probing the servers and inspecting the packet headers and their reaction to emulated network events (e.g., packet loss). Please refer to § 5.10.1 for more details about the tool. Using the tool, we fingerprinted the configurations for the Alexa top 1k websites from five different regions (North America (N.A), South America, Asia, Europe, and Australia), and present the results in Table 5.2. We use N.A configurations as the reference point and compare the observed configurations along two axes:

**Observation 1: Heterogeneity across CDNs:** In Column 3 (cross-CDN), we observe that different CDNs use different configurations in N.A. While some of the heterogeneity can be attributed to differences in the default values for different OSes, we observe that CDNs do use non-default values, e.g., amazon.com uses an ICW of 24 MSS in N.A.

**Observation 2: Homogeneity within a CDN:** In Column 4 (cross-region), we observe that only a small number of CDNs tune their network stack to account for regional differences, i.e., use different configurations in N.A. than the other regions. The highest amount of tuning occurs at L4, with 6.9% of the CDNs tuning the ICW differently in N.A. than in other regions, e.g., 24 MSS in N.A. but 10 MSS in Asia for amazon.de.

---

1\[https://systems.cs.brown.edu/projects/configtron/\]
**Takeaway:** Taken together, these observations indicate that while individual CDNs perform modest tuning, most do not tune finely enough to account for regional diversity. In fact, only a small set of CDNs configure differently across regions.

### 5.2.2 Implications of Configuration Tuning

Next, we quantify the benefits of dynamically tuning a web server’s networking stack by conducting a large scale study in our local testbed. We emulate a wide range of representative networks (extracted from real-world traces [181, 89, 45, 543]) and perform an exhaustive, brute-force search of configuration space (detailed description of the traces is provided in § 5.6.1). Table 5.1 lists the set of configurations, with default settings for TCP and HTTP taken from the Linux transport stack (kernel 4.20) and Apache (v2.4.18), respectively. In each trial, the server iteratively selects a configuration from the possible configuration space, a representative network is emulated using NetEM [223], and the PLT of a randomly selected website from Alexa Top-100 (locally cloned on the server) is measured five times. The optimal configuration is defined as the one that results in the lowest PLT for a specific network and website.

Figure 5.1 explores the implications of using sub-optimal configurations, by comparing optimal and default configurations for pageloads across diverse networks and websites. We observe that there is $\sim 18\%$ PLT improvement at the median (over 70% at tail) when optimal configurations are used over the default. While the number may appear small, they can result in tremendous revenue improvements [78, 145], and more in the developing regions where CSPs are investing heavily to improve network [174, 316]. We observe the highest reconfiguration benefits for low bandwidth, high RTT/loss regions, representative of developing region networks.

Next, we analyze congestion control measurements across different regions from Pantheon [543]. We observe that emerging protocols, e.g., BBR, PCC, or Remy, which use probing or ML to improve performance, do not provide uniformly superior performance. In particular, we observed that in many situations BBR is suboptimal, performing 3X to 10X worse than the optimal congestion control. Moreover, no congestion control is optimal for more than 25% of the networks tested, and the median congestion control is optimal for only 6% of the networks.

### 5.3 Configanator’s Algorithm

Tuning network configurations to maximize the web performance for diverse networks and end-users presents a complex learning problem. Next, we formulate the problem and present a domain-specific ensemble to address the challenges.

**Problem Formulation:** Given a set of networking configurations ($C=\{c_1, c_2...c_n\}$), network conditions ($N = \{n_1, n_2...n_n\}$), devices ($D = \{d_1, d_2...d_n\}$), websites ($W = \{w_1, w_2...w_n\}$) and a function, $f()$, that maps a website, network condition, device, and configuration to a metric of web page performance (e.g., PLT or SpeedIndex). Note that, $f(c_i, n_i, d_i, w_i)$ returns the web page performance metric value for applying configuration $c_i$ to a user device $d_i$ loading website $w_i$ in network $n_i$. In this paper, we use PLT as the metric for web page performance and can be easily replaced with other metrics. Our goal is to solve Eq. 5.1 and find a configuration ($c^*$) that minimizes $f$ for a given combination of $n_i$, $d_i$ and $w_i$. 
Solving the black-box function \( f() \) requires exploring sample space. Two possible exploration algorithms are:

- **Brute force** [25] which tests each possible configuration one by one until the entire space is explored.
- **Bayesian optimization** (BO) [404, 79] is a principled global optimization strategy that uses a prior probability function to capture the relationship between the objective function (Eq. 5.1) and the observed data samples. BO models \( f(c,n,d,w) \) as a Gaussian process (GP) [79]. GP is a distribution of candidate objective functions and is used to select the next promising point \( (c^*) \) which is then evaluated on a connection. GP then updates its posterior belief by adding the new observation \( f(c^*, n, d, w) \) to the set of seen observations. With every new observation, the space of possible candidate functions gets smaller and the prior gets consolidated with the new evidence.

**Challenges:** Both approaches are sub-optimal for our use-case due to several reasons: (1) non-stationary network conditions [554, 57, 255, 258] (network conditions change every few minutes), (2) BO assumes that data is noise-free or only has Gaussian noise [458], and non-Gaussian noise (tail latency can not be modeled by a Gaussian process [310]) disrupts the estimation of next candidate sample and is observed to impact BO’s hyper-parameters (e.g., threshold on expected improvement for next sample to stop the exploration), (3) costly data collection (collecting data requires testing on end-users which can impacts PLT and revenue), (4) data scarcity (testing on individual users requires each user to generate a tremendous number of connections but a user may only visit the site a few times).

**Intuition:** The intuition behind Configanator’s algorithm is to decompose the model building into two phases: (i) an initial phase during which the search should be directed to speed up the process and build a good (not perfect) model, and (ii) a steady-state during which the search should be more stochastic to iteratively improve the model and tackle non-Gaussian noise. Building on these insights, Configanator leverages a combination of clustering, an ensemble of bandit-techniques, and ML to address the aforementioned challenges. Specifically, clustering is used to group connections based on their network and device similarity (called *Network Class*) and aggregate observations across similar connections to address data scarcity. The use of a contextual multi-armed bandit [513] enables Configanator to explore configurations and continuously collect data samples to learn and tackle dynamic client-side conditions in a balanced and online manner. To generalize observations across the connections, a Decision Tree is trained for efficient inference.

### 5.3.1 Domain-Specific Multi-Armed Bandit

Configanator’s learning algorithm consists of a contextual multi-armed bandit [513, 322, 303] with three arms:

- **Exploration Arm-1 (Gaussian process [404, 435]):** The Gaussian process (GP) bandit [29, 287] uses an acquisition function to perform a directed search to quickly discover a “good” (might not be optimal) solution when no information exists for a Network Class (NC). There are multiple acquisition functions available [79] and we use Expected Improvement (EI) [435] because of its well-documented success [29, 198, 151]. This
The search process includes two terminating conditions: a threshold on EI and minimum of number of data points to explore. For non-continuous configurations (e.g., HTTP version), we encode them into a number to discretize the space\(^2\). To account for performance differences between websites and NCs, the GP-arm is composed of a collection of GP models, one for each unique website and NC combination (§ 5.10.4).

- **Exploration Arm-2 (Epsilon-bandit [500]):** The Epsilon-bandit randomly re-samples the data points to overcome issues endemic with the Gaussian process (and Bayesian Optimization in general), e.g., non-stationarity of mean performance. The network operator bounds the random exploration by defining a parameter, \( \epsilon \), that controls the trade-off between speed of exploration and the impact on end-user QoE. A high \( \epsilon \) improves exploration but results in a negative impact on clients’ QoE due to constantly changing configurations.

- **Exploitation Arm (Decision Trees [417]):** The exploitation arm uses ML-powered prediction to model the data collected through the exploration arms. We evaluated several techniques including Support Vector Machines, Decision Trees (D-Trees), and Random Forests. We found that the D-Tree hits the sweet spot, providing comparable accuracy to the other models while being efficient enough to build and update at scale. The D-Tree encompasses all websites and NCs to learn across websites and networks. Leveraging the config-performance curves collected by underlying exploitation arms, a single D-Tree model is trained for the “good”

\(^2\)GPyOpt [390] supports mixed (continuous/discrete) domain space [391].
configuration found so far for each website/NC pair, and the D-Tree maps \{website, device, network/AS characteristics\} to their optimal configuration.

**Context-based arm switching:** Configanator constantly switches between the arms based on the NC’s “context” which is defined as the quality of the GP-model for the website/NC. It operates in two modes: (i) **Bootstrap**, when no information exists for a website or NC, the context is empty and the GP-arm is used to explore the configuration space in a principled manner until the acquisition function (EI) indicates that a good configuration is found, (ii) **steady-state**, when information from the GP-arm indicates “good” configurations, Configanator uses either the epsilon-bandit to further explore the configuration space, or the exploitation arm (i.e., D-Tree) to leverage best configurations. Note that, random exploration through epsilon-bandit continues after EI threshold is met.

### 5.3.2 Discovering Network Classes

**Configanator** extends on observations from prior studies \[255, 361\] and classifies homogeneous connections into Network Classes (NC) with the intuition that similar connection characteristics lead to identical optimal configurations.

**Design Goals and NC Features:** The ideal NC-clustering should (i) create a small number of clusters, each with a large number of connections to amortize the cost of explorations, and (ii) all members of a cluster should have near-identical profiles. The two goals inherently contradict: the greater the number of entities in an NCs, the higher the probability that the NC contains entities with diverging performance. The second goal is further complicated by the sensitivity of a configuration’s performance (e.g., PLT) to a myriad of factors in the end-to-end connection. To this end, we use network characteristics (bandwidth, latency, loss rate), AS information (ASN, geo-location), and device type as the basis for measuring similarity.

**Capturing NC Features:** To enable Configanator to effectively tune both the transport and HTTP layers, we must identify all features during the TCP handshake before the HTTP version is negotiated through ALPN \[243\]. If we identify features after HTTP negotiations, then tuning the HTTP layer would require renegotiation and hence incurs latency penalty. In Figure 5.2, we highlight the features collected during specific phases of the connection: (1) During the TCP handshake, we capture RTT, IP-prefix, and ASN/geo-location\(^3\). (2) During the TLS handshake, we apply TLS fingerprinting techniques \[30, 260, 81, 493\] on the TLS *Client Hello* to perform device identification and capture device features (accuracy evaluated in § 5.10.2). Note that, most operators already employ TLS fingerprinting for security purposes \[484, 244, 38\] and is also supported by major web servers \[144\]. We use the *Server Name Indication (SNI)* in the *Client Hello* to determine the website hostname which is one of the input features for the learning framework. (3) For goodput and loss rates, features that cannot be captured during handshake, we build and use a historical archive of these network characteristics.

**Network Classification:** Clustering can be done using conventional techniques, e.g., K-means, hierarchical, or domain-specific techniques \[80, 176, 408\], e.g., Hobbit \[297\] or, CFA \[255\], or using CDN state of the art \[101, 462, 530, 334, 531\], e.g., latency-based groups \[101, 462, 530\]. Although Configanator

\(^3\)Captured using end-user’s IP and publicly available data (RouteViews for AS \[80\], MaxMind for geo-location \[245\])
can incorporate any of the aforementioned techniques, our prototype uses “K-means” clustering because of its simplicity. Configanator empirically selects the smallest K (i.e., the number of classes) that bounds the spread of performance within each NC by a predefined limit\(^4\)(evaluated in § 5.6.2 and § 5.10.4).

### 5.3.3 Configanator Workflow

Figure 5.3 presents the end-to-end workflow. Default configuration is initially used for a newly-seen IP-prefix (①, ②) due to the lack of information about its goodput and loss-rates. For any subsequent connection from the IP-prefix, the recorded, as well as the actively collected features, are used for NC classification (③). If the network, AS and device characteristics do not fit into an existing NC, a new NC is created (④) and the next init_samples connections for the respective NC are used for bootstrapping (⑤) its empty context. When the respective NC is bootstrapped, the multi-armed bandit uses the actively and passively collected features, as well as the requested website, for determining the context and alternates between the arms (⑥). The connections are correspondingly tuned (⑦) and the resulting performance metrics are fed back into the models to help refine their classifications and improve accuracy. Due to the computationally intensive nature, Configanator builds/updates NC clusters in the background and uses the already-built clusters for real-time classification.

### 5.4 Architecture

Our re-architected web server consists of four components (Figure 5.4): The HTTP server application [468, 467, 382, 510] operates as it does today: serves content and collects performance metrics for each connection. The Configuration Manager runs the learning algorithm on the telemetry collected from the web servers. The Configanator-API abstracts vendor-specific configuration details and provides a uniform interface for configuring web server’s network stack parameters. A Configuration Agent runs on each web server and uses the information received from the Configuration Manager to configure the connections through the Configanator-API.

Adopting this architecture in an incrementally deployable manner is practically challenging. The configuration parameters are exposed in an ad-hoc manner, e.g., tuning transport configuration requires IOCTL and setsockopt, while tuning HTTP requires changes to application code and enhancements to the ALPN protocol. Additionally, most CDNs use well-established code bases and exposing the configuration interfaces required by Configanator should incrementally build on the existing code.

#### 5.4.1 Configanator-API

The Configanator-API presents a uniform interface over the web server’s serving stack thus abstracting away OS and web server specific details. This simplified interface enables the Configuration Agent to easily tune the network stack, without having to understand vendor-specific details or implications.

\(^4\)Controlled by NCSpread knob in simulator (Table 5.4 in § 5.10).
**Transport tuning:** Unfortunately, the traditional kernels only expose and provide flexible reconfiguration for a subset of TCP’s parameters. In particular, some parameters (e.g., ICW) can be configured on the connection level, while others can only be configured on a global scale (e.g., tcp_low_latency). Using Configanator at a coarser granularity, either limits the type of supported connections on a machine or limits the configuration space. There are several options to address this issue ranging from user-space TCP/IP stacks [155, 252, 409], kernel modules, eBPF programs, to leveraging virtualization. We opt for a kernel module-based design over virtualization approaches because hosting a single configuration per VM introduces significant overheads.

**HTTP tuning:** HTTP version and H2 settings are determined through Application Layer Protocol Negotiation (ALPN) [243] in TLS handshake and H2 SETTINGS [63] frame, respectively. Given the requirement for per-connection tuning, we augment the ALPN and the H2 SETTINGS frame code to enable fine-grained control over these configurations. In particular, Configanator configures these settings by restricting the options presented in the server advertisement to the configuration setting being tuned, e.g., to set the HTTP protocol to H2, we limit the “ALPN next protocol” field in TLS Server Hello to just H2. Similarly, we restrict the options in the SETTINGS frame to configure HTTP/2 settings.

**Tuning Workflow:** Configanator-API tunes both the TCP and HTTP version during the TLS handshake: after receiving the Client Hello from the end-user and prior to sending the Server Hello. This is the perfect location to tune because (1) the complete feature set required to determine a connection’s NC and configuration can be captured at this point, and (2) the server is yet to finalize the HTTP protocol, which the ALPN selects in Server Hello, thus enabling us to configure the HTTP version. We note that at this phase of the connection, the TCP state machine is in its infancy because the sender has not sent any data, and thus virtually no significant state is lost when we change the congestion control algorithm or settings.

### 5.4.2 Configuration Agent

The Configuration Agent is the glue logic between the Configuration Manager and Configanator-API — it collects the connection features, uses rules provided by the Configuration Manager to make configuration decisions, and configures them using the Configanator-API. We select a proactive approach, where the Configuration Manager constantly pushes NC and configuration mappings to the Configuration Agent which caches them locally. Further for an unseen IP-prefix, Configuration Agent uses the default configuration, until the Configuration Manager finds a better mapping.

### 5.4.3 Configuration Manager

The manager runs in a centralized location, e.g., a data center or locally in a Point of Presence (PoP), with the implications later explored in § 5.10.6. It is charged with running the learning algorithms (§ 5.3), network classification models (§ 5.3.2), and disseminating the configuration maps to the Configuration Agents’ cache. The Configuration Manager disseminates and collects data from the Configuration Agents using distributed asynchronous communication. For the NC and configuration maps, Configuration Manager broadcasts to all Configuration Agents, whereas for reporting performance data and for making one-off-request for configuration maps, the Configuration Agents use unicast.
5.5 Prototype

The implementation highlights of the prototype are as follows:

**Configanator-API:** partly resides within the kernel (as a module) and partially resides in user-space in the form of additions to the web server code (in our case Apache). The components within the kernel allow us to tune the transport, while the user-space allows us to tune the HTTP layer.

The kernel module reuses functions provided by kernel’s congestion controls through the `tcp_congestion_ops` interface and tunes fields in appropriate structs (e.g., `inet_connection_sock`). For tuning globally-defined knobs at a per-connection level (e.g., `tcp_low_latency`), we leveraged kernel patches [153, 226] to define and reference them from `tcp_sock` struct. The user-space component within Apache code tunes HTTP version in Apache and its design is generalizable to other servers that use OpenSSL. OpenSSL library is used by most web server implementations and allows web servers to register a `SSL_CTX_set_alpn_select_cb` [191] callback to modify ALPN decisions. To tune HTTP version, we register a callback which looks up the HTTP version to use for a connection and restricts the ALPN options advertised to the one specified by the Configuration Agent. For H2 settings, we modify the Apache H2 module to dynamically select the configurations while sending the SETTINGS frame. The user-space agent also generates the TLS fingerprint for device identification. We use JA3 [260] for TLS fingerprinting. In both our testbed experiments and in the live deployments, we use the ALPN-centric approach which modifies protocol options presented in the advertisements.

**Configuration Agent:** is user-space code and is implemented in 492 LoC of C++ code. The agent updates TCP and HTTP settings via the Configanator-API. This component also parses Apache’s logs for measuring network characteristics. For measuring the PLT, the web server injects a simple JavaScript into the webpage to measure the navigation timings.

**Configuration Manager:** is developed in 1435 LoC (Python). It uses SciLearn [447] for D-Tree and GPyOpt [390] for the Gaussian Process. For communication with the Configuration Agents, we use ZeroMQ [548]. For D-Tree, we use SciLearn’s CART algorithm with the following configuration: (i) entropy for the information gain, (ii) set the minimum number of leaf nodes to 80, (iii) set the minimum number of samples needed for the split to 2, and (iv) do not limit the depth of tree. For Gaussian process, we use `init_sample=4, min_sample_tested=7` and `EI=8%` thresholds. We tested a range of these hyper-parameters and selected the ones resulting in the highest accuracy. Following [29], we tested EI threshold in 3-15% range and selected 8% for its best trade-off between accuracy and search cost. For controlling the “K” for NCs, we use a `NCSpread` threshold of 5% (§ 5.10.4).

5.6 Evaluation

We evaluated Configanator through a large-scale, trace-driven simulator using real-world traces, and live-deployments (§ 5.7). The simulation enables us to understand the system behavior under dynamic conditions, as well as analyze the implications of individual design choices.

---

5(e.g., entropy vs Gini impurity for information gain, number of leaf nodes ranging from 50 to 500, ID3, C4.5, and CART for D-Tree)
Datasets: To simulate client activity, we use data from five sources: (i) GlobalCDN comprises 8.2M requests sampled from web and video services from 3 GlobalCDN PoPs (two in N. America, one in Europe) for a duration of 6 hours. Each request is a client fetching an object (e.g., web object, video chunk, etc.) and contains user information (IP prefix, ASN, etc.), observed server metrics (goodput, RTTs, loss rates etc.), CDN logs (e.g., user to edge PoP mapping [462, 101]) and performance metrics (time-to-last-byte). (ii) CAIDA [89], packet traces from the Equinix data-center in Chicago (in 2016). (iii) MAWI [45], packet traces from the WIDE backbone in Japan (in 2017). (iv) FCC [181], a U.S. nation-wide home broadband dataset. (v) Pantheon [543, 542], a data set of client sessions across different regions.

Generating client sessions: We use our traces to characterize the network conditions of real-world users. CAIDA and MAWI traces are captured at a vantage point between the client and server and we measure the goodput, RTT and loss rate by sequence-matching the data packets with their ACKs. Measurements are on a per-request granularity. For multiple readings within the 5s window, we aggregate and use the median value.

---

6Over a 5-second window (tunable through ChunkSize parameter in the simulator), e.g., data ACKed in 5s is used to measure goodput between vantage point/user. Further, we ignore duplicate ACKs while measuring RTTs.

7Measurements are on a per-request granularity. For multiple readings within the 5s window, we aggregate and use the median value.
end-points.

**Configuration Rewards:** To avoid the pitfalls of trace-driven simulations [61], we decouple the modeling of configuration rewards (i.e., PLT calculation) from the process of generating client sessions. Our testbed comprises a cluster of 16 Linux servers (kernel 4.20), divided evenly to act as server (Apache) and clients (Chromium [211]). Our control over the machines and network enable us to set arbitrary server-side configurations (from Table 5.1) and emulate the bottleneck link to match the measured goodput, latency and loss rates from the datasets (using NetEm, TC [223]), with buffer-size set to Bandwidth Delay Product (BDP). To isolate the impact of network and configuration on PLT, caching (server or browser) is disabled and each server serves a single client (no resource contention). Using this testbed, we exhaustively measure the PLT for all combinations of configurations (Table 5.1). For each \{network condition, configuration\} pair, each website is loaded multiple times with the browser⁸. The final results are stored in a large tensor that maps \{network condition (goodput, RTT, loss-rate), configuration, website\} to PLT – called the PLT-Tensor comprising data from the pageloads in the testbed.

**Simulator (Virtual Browser):** Leveraging the client sessions and the PLT-Tensor, the simulator simulates the client’s browsing behavior and interaction with Configanator as follows (visualized in Appendix 5.10.6): (i) website⁹, user information (e.g., IP) and session characteristics are taken as inputs, (ii) the learning framework determines the appropriate configuration for a connection, and (iii) pageload is simulated by using the PLT-Tensor to determine PLT for the client given the selected configuration. The simulator feeds the PLT back to the learning framework to complete the feedback loop.

PLT is sensitive to a myriad of features, ranging from dynamic network conditions at different time-of-the-day [255], user devices, to inherent variability. The session time-series captures the network dynamics and the testbed isolates the impact of network conditions on configurations. Table 5.4 lists the set of knobs we leveraged to test various realistic design choices, e.g., NCSpread to test various “K” sizes, PerfMemory to test the impact of PLT variability, etc. To account for the other factors like user devices, we conducted a scaled-down experiment on a CDN and tested different configurations for the real-world, diverse user devices (results in § 5.7.1).

**Alternate algorithms:** We evaluate against 8 algorithms:

**(i, ii) Brute-force (Brute, Brute+NC):** explores individual configurations, in an online manner, until all are explored and uses the best one (i.e., lowest PLT) for subsequent connections. Brute learns at the granularity of individual clients (i.e., unique IP) while Brute+NC clusters clients into Network Class (NC), and thus learning is spread across each NC.

**(iii, iv, v) Bayesian Optimization (BO, BO+NC, CherryPick+NC):** Bayesian Optimization is used to explore the configuration space and the best-explored option is exploited once BO-specific thresholds are met (§ 5.5). BO learns per client and BO+NC learns on a user group (NC) granularity. CherryPick+NC is similar to BO+NC but with hyper-parameters specified in [29].

**(vi) Multi-armed Bandit (MAB+NC):** uses traditional MAB with a weighted epsilon-greedy agent [513]. Each arm of the bandit is a different configuration, tested on NC granularity.

---

⁸We repeated each measurement 5 times, similar to [520].

⁹We iteratively load every website from our corpus for a given session.
(vii) **Random**: Randomly selects a configuration in each trial.

(viii) **Optimal**: An oracle suggests the optimal parameters for a session by offline brute-force, i.e., PLT is calculated for the entire configuration space for each session offline and the configuration with the lowest PLT is used. This process is repeated for every session and puts an upper bound on improvement.

### 5.6.2 Effectiveness of Configanator

Figures 5.5, 5.6 present the improvement in PLTs over default configurations for the different algorithms. The box plots compile data across the website pageloads for the client sessions in the respective trace. Configanator outperforms all alternatives at median and tail, improving p95 PLTs by 67% for GlobalCDN (1500ms), 36% (1100ms) for MAWI, 32% (610ms) for FCC, 48% (640ms) for CAIDA and 57% for Pantheon (850ms). Unlike Default, while Brute and BO apply different configurations to users, they assume that the network remains static and are unable to adapt to fluctuations. Moreover, due to its inability to adjust to fluctuations, BO often explores over 90% of the space without achieving the target EI, behaving similarly to Brute. Brute+NC, BO+NC and CherryPick+NC improve over the prior by amortizing the costs of learning but fail to adjust to non-Gaussian variations. Although MAB+NC is able to handle non-Gaussian noise, it explores/exploits on a per-NC basis and, due to the lack of a cross-NC exploitation arm (Configanator’s DT), MAB+NC falls short in its ability to apply patterns learnt across NCs.

As Configanator continuously learns and tests new configurations in an online fashion, a ‘bad’ configuration may be tested during the exploration phase and may lead to performance degradation. This behavior contributes to the worse PLT than Default for the p5 pageload in Figures 5.5, 5.6. A breakdown of the performance degradation and its causes are presented in § 5.10.6.

**Dissecting Performance Improvements**: Next, in Figure 5.8, we analyze performance breakdown for a subset of websites according to the networking conditions used in prior work [520]). We make two observations: (i) Improvements tend to be higher in low bandwidth, low to high RTT/loss networks (typical for developing regions) with a median value of 14-67% compared to 10-25% for high bandwidth. We postulate that this trend is an outcome of the higher focus on developed region networks (typically high bandwidth, low RTT/loss) for the default configuration selection [152]. We observe a similar trend across our traces: GlobalCDN, MAWI and Pantheon traces (p95 RTT in 100-180ms) tend to show higher improvements than FCC and CAIDA (US-based, ~60ms p95 RTT). (ii) the websites with highest benefits tend to be content-rich, e.g., 9gag.com and cnn.com observe >45% and >60% improvement, respectively, for all low bandwidth networks.

### 5.6.3 Benefits of Learning Ensemble

Next, we analyze the convergence for the top-3 algorithms from § 5.6.2 to focus on the aspects of Configanator that lead to better performance. We further split Configanator into two versions to analyze the benefits of its bandits: “NoGP” lacks GP and guided exploration, while “NoDT” lacks the decision tree.

Figures 5.10 plots the median distance from optimal across all NC and websites, for the first 500 update iterations. The observations are: (i) As data is gathered, Configanator performs better than others because of its ability to blend the benefits of both GP and DT – essentially efficient exploration and effective exploitation.
Figure 5.10: Cold-start convergence to optimal across NCs.

Figure 5.11: Impact of number of NCs (K) on performance.

(iterations 3-10). Brute+NC exhaustively explores the complete space before converging to a choice, while MAB+NC exploration lacks the guided nature of acquisition function. (ii) Eventually, with sufficient data Configanator-NoGP is able to use the decision tree’s predictive power to achieve near ideal performance (iterations 100+). Although MAB+NC gets within 2-3% of optimal for these iterations, it still needs more iterations to reach the optimal. (iii) While NoGP perform comparably for median at 200+ iterations, performance at the tail is still different (§ 5.10.6).

<table>
<thead>
<tr>
<th>NC group</th>
<th>PLT ratio</th>
<th>Prefix distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=5</td>
<td></td>
<td>p5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.92</td>
</tr>
</tbody>
</table>

| >=30     |           | 1.00 | 1.03 | 1.28 | 1.37 | 1.33         |

|          |           | 0.89 | 0.89 | 0.89 | 0.89 | 1.08         |

|          |           | 1.08 | 1.09 | 1.17 | 1.24 | 2.67         |

Table 5.3: Impact of NC size on performance.

5.6.4 Impact of Network Classes

Impact of Number of NCs: Next, we evaluate the impact of our clustering configuration (i.e., NCSpread) and analyze how the cluster size impacts performance. Intuitively, NCSpread bounds the performance

Figure 5.12: Impact vs TCP connection reuse.
variance within a cluster and has a direct impact on the number of clusters, or ‘K’. Given a NC\textsuperscript{Spread} value, the simulator performs a brute-force search to determine the smallest K that yields the threshold. We tested three scenarios with K inflated to \{1.5, 2, 3\} times the baseline value (Figure 5.9 experiments). The inversion from NC\textsuperscript{Spread} to K and its implications on modeling accuracy are further discussed in § 5.10.4.

Figure 5.11 plots the ratio of Configanator and \{MAB+NC, Brute+NC\} PLT across the pageloads in GlobalCDN trace (\(< 1\) when Configanator outperforms). We observe the performance gap between Configanator and others increases with the K size. Although the large K results in a higher number of tighter NCs with lower performance spread within their constituents, it leads to an overall increase in exploration steps for MAB+NC and Brute+NC, as these algorithms explore the individual NCs independently. Further, the individual NC’s best-found configuration is exploited for a narrower set of connections due to a lower number of connections in each cluster as compared to the case when K is small. On the other hand, the DT-arm in Configanator builds on the data collected for all NCs (§ 5.3.1). As soon as Configanator switches to DT-arm fairly early (§ 5.10.6), it is able to exploit the best-found configuration for a wider audience, irrespective of the NC boundaries. The higher degree of exploration required by MAB+NC and Brute+NC makes their performance sub-optimal for the NCs with a smaller number of connections. Moreover, this can also lead to performance problems for tail connections, who are often in smaller NC due to their divergent network and device characteristics.

**Impact of Size of NC (# of connections):** Configanator aggregates network measurement across similar connections and assumes homogeneity within an NC. Though an NC with a small number of users may lead to a smaller number of connections to learn from, it also favors the system as connections in the respective NC are strictly homogeneous. Next, we explore the impact of this bias on our results. We divide the NCs based on their unique number of IP prefixes and compare the PLTs observed for the individual prefixes with the NC’s global PLT, i.e, median across all the prefixes in the NC. For two of such divisions, Table 5.3 presents the PLT comparison across the prefixes in NC groups. Compared to the \(<5\) group, i.e., NCs with a small number of distinct prefixes, where performance for most prefixes matches the global one; \(>=30\) group shows more varying performance (e.g., lower than global PLT for the p25 prefix). However, we observe that the presence of larger NCs does not drastically impact Configanator as performance for most of the prefixes is still on par with the global one. For the tail prefixes that performs poorly as compared to the global PLT for
the $\geq 30$ group, Configanator overfits the best-found configuration for the NC majority to the tail prefixes, and is observed to still outperform the Default (row 4 and 6 in Table 5.3).

5.6.5 TCP Connection Reuse (ConnReuse)

CDNs typically employ ConnReuse, allowing a new request to reuse older TCP connection. The key advantage of this feature is that the new request inherits matured congestion window ($cwnd$) and does not restart the connection from scratch, i.e. ICW. To analyze connection reuse, we analyzed the trace (GlobalCDN) to identify if and when requests reused existing connections and modified our setup to employ the reused connection’s $cwnd$ as the ICW for the page load\(^{10}\).

Figure 5.12 plots Configanator improvement over ConnReuse. We observe that Configanator gains are reduced from 18% over Default (Figure 5.6) to 14% at the median. The benefits at the tail are still substantial, with 56% p95 improvement. There are several reasons for this behavior: First, connection reuse only impacts the slow-start phase (e.g., ICW) and does not tune the CCA and HTTP, the top two critical knobs (Figure 5.17). Second, even with reuse, a connection is not always guaranteed to reuse the old $cwnd$, since other TCP settings like slow_start_after_idle may reset to default ICW — forcing a reused connection to again go through slow start phase. In fact, the old $cwnd$ is reused with a probability of 0.27 in our trace, i.e., only a small subset of requests exploit the benefits of reuse. Third, unlike Configanator’s exploitation of good configurations for similar connections, the scope of ConnReuse if limited to a single connection — a new connection from even the same user will go through the default slow start phase. Consequently, while connection reuse outperforms the Default by only 4.65% and 19.6% at median and tail respectively, a variant of Configanator that only tunes ICW still performs ConnReuse with 8.7% and 23.4% improvements at median and tail.

These results suggest that ConnReuse alone is not the silver bullet, also portrayed by other ICW tuning system [187], and Configanator is expected to bring substantial improvements even when traditional optimizations are considered.

5.6.6 System Benchmarks

Next, we evaluate the latency, CPU and memory overheads. The experiments are performed in a testbed by emulating network conditions from our traces for 10 randomly selected websites. We repeat each test 1000 times.

**Latency Overheads:** For latency overheads, we focus on the modifications to ALPN to enable HTTP level tuning. We compare Configanator against a version that does not modify ALPN and tunes HTTP level by renegotiates which incurs at least 1-RTT overhead. Figure 5.15 plots the PLT for the two variants, normalized by Default (vanilla Apache). Given that Configanator simply edits the “ALPN next protocol” field in TLS ServerHello without requiring any extra communication, we observe no latency overheads and a similar performance to the Default. For Renegotiation, we observe a slight PLT inflation ($\sim 3\%$ at the median) which is due to the TLS renegotiation required to switch the HTTP version. We note that this

\(^{10}\)We infer ConnReuse if the first $cwnd$ for a request is greater than connection’s ICW. Our GlobalCDN trace directly captures these fields for each request. Note that, the reused connection may also inherit the MTU and SRTT values, but we limit our focus to the key component that limit data transfer, i.e., $cwnd$. 
approach still has a minor overhead (4% higher PLT at median) because a page load requires many RTTs and this overhead get amortized.

**CPU and Memory Overheads** To measure the CPU and memory overheads, we leveraged the Apache Benchmark tool to setup 100, 250 and 500 concurrent connections. We observe slight CPU overhead (< 5%) as compared to Default. Although reconfiguring the connections do not require any additional memory, keeping the IP prefixes and their NC/configuration rules in the KV-store contributed to an increased memory usage.

### 5.6.7 Fairness Implications

Next, we explore the fairness implications. Within the testbed, we explore the situation where 30 concurrent flows share a representative bottleneck link, i.e., the access links for 3G, 4G, etc (number of flows from [446] § 5.10.6), under shallow buffers ({0.5 and 1} BDP). We use Jain’s Fairness Index [249] to quantify fairness. We split the connections into two groups – one using Configanator and another using the default configuration (e.g., Cubic with 10MSS ICW). We then vary the percentage of connections in each group.

**Quantifying Unfairness:** Figure 5.13(a) present Jain’s index when 75% and 50% of the flows are tuned. We observe that fairness decreases as the percentage of Configanator-tuned flows increases. Unsurprisingly, unfairness arises for two reasons: (1) when a flow is configured to use BBR [433, 229, 501, 523, 96], and (2) when a flow is configured to use high ICW values (even if BBR is not used) [286, 352, 214].

**Configanator without unfair configurations:** Next, we excluded the unfair configurations from the
configuration space and tested 3 scenarios: (i) prevent BBR usage, (ii) prevent high ICW usage, (iii) prevent both. Figure 5.14 plots the ratio of PLT seen for vanilla Configanator (all configuration) to the variants, for GlobalCDN traces. We observe that NoBBR and NoHighICW perform similar to vanilla system for a significant fraction of the trace (63% and 35% respectively) and within 6% for worst case: this is because BBR is not always the optimal choice and application layer tuning (HTTP version) helps account for the lack of BBR or HighICW.

The results show that Configanator can provide an alternate war-chest to CDNs to improve web performance, even without using the unfair configurations.

5.6.8 Critical Knobs

We analyze the relative importance of reconfiguring different configuration parameters (Table 5.1). Our goal is to understand the minimal (or critical) parameters that must be tuned to significantly improve performance. In Figure 5.17, we plot the performance benefits of using distinct subset of configuration parameters, leveraging the brute-force exploration data from PLT-Tensor. We observe that the top 3 crucial parameters are HTTP version, congestion control algorithm (TCP-CC) and ICW. Moreover, when performing a layer to layer comparison, we observe that the Transport layer parameters combined (Tran. layer) have a higher impact on performance than the Application layer knobs combined(App layer). To explain this discrepancy, we analyze the different knobs in each layer and we observed that while certain transport knobs, e.g., Auto Corking, have little benefit in the median scenario, they are influential at the tails. Unlike the transport layer, in the Application layer most of the parameters (e.g., HTTP2 settings like header table size etc.) do not show significant benefit in median or tail conditions. We further analyzed the connection and request features that had the highest impact on the tuning for a certain knob. Interestingly, the website parameter has the strongest impact on the choice of HTTP version, while the network properties (goodput, latency and packet loss rates) strongly impact the choice of TCP configurations. HTTP version is sensitive to the website parameters as different websites differ in the content complexity (e.g., number and size of objects) and the performance for the different HTTP versions vary for the different content complexities [520]. On the other hand, different TCP CC model different aspects of networks, e.g., react to packet loss or latency inflation, and that is why we observe the network properties to be the key factor in the selection of TCP configurations.
5.7 Live Deployment

In this section, we present the results for dynamically tuning the configurations at scale through a controlled experiment at GlobalCDN and a live prototype deployment on Google Cloud with 3161 end-users.

5.7.1 Validation at GlobalCDN

Next, we validate our approach when applied to data with more realistic and diverse client settings. We conducted measurements at GlobalCDN to collect data regarding performance of different configurations for the diverse networks and end-user devices. Specifically, we used the default configuration for 80% of the connections and explored random configurations for the rest to generate the data needed to emulate the contextual multi-armed bandit based exploration. Due to operational constraints, we analyzed a subset of configuration knobs: different congestion control algorithms and ICW. The experiments were conducted by randomly selecting 1% of the users from 3 of the CDN PoPs, for a duration of 6 hours. Note, these PoPs have the same workloads as the GlobalCDN trace described earlier.

We replayed the captured traces in our simulator, with two key distinctions: (i) the testbed-based PLT-Tensor was replaced by TTLB measurements collected from production users, since these TTLB measurements encompass the performance across real-world users, (ii) This experiment covers diverse user devices in-the-wild. Figure 5.16 presents the TTLB improvements for Configanator (versus default configuration) with upto 37% improvement at the tail (p95). Although this simulation covers a smaller configuration space, the improvements affirm the efficacy of tuning at scale, working with the diverse set of NC features (user device, network, geo-location, AS).

5.7.2 Google Cloud Deployment

We deployed Configanator on several Google Cloud servers, each with 8 CPU cores and 32 GB of RAM. We evenly divide the servers into two groups: one half with the Configanator-enhanced servers, while the other half with traditional Apache server. We cloned a variety of real-world websites from Alexa top-100 and hosted them on servers without sharding. We hosted the Configuration Manager on a dedicated instance.

For clients, we used SpeedChecker [319, 320], a platform for global Internet measurements with vantage points deployed across the globe. We had 3161 clients in total, spread across 4 of the continents. The clients periodically conducted pageloads from both the Configanator and the traditional web servers at the same frequency, resulting in ~150K pageloads in 21 days. Further details about SpeedChecker are provided in § 5.10.6.

Figure 5.18 plots the raw PLTs observed for the two systems, with the accompanying table summarizing the PLT difference and improvements. Due to the online nature of the exploration and learning, we observe PLT degradation for a small subset of pageloads: 4.3% of the pageloads faced upto -13% degradation. For the rest, Configanator resulted in significant improvements, with upto 3.8s improved PLT at the tail (upto 767ms for the median). Dissecting the improvements across networks and websites, we observe a trend similar to Figure 5.8: low bandwidth, high RTT/loss networks and content-rich websites get the most benefits. For the top configurations, we observe no clear winner: top 5 covered 3 CCs (BRR, Cubic and Vegas), both HTTP
versions, and ICW ranging from 16 to 40. We observe a stark difference in the ICW values used by clients in developed regions (Europe, N.America), with higher ICW (30-50 MSS), compared to developing regions (16-24 MSS).

Most of the clients (∼75%) are from N.America/Europe and the rest are geographically distributed which results in unbalanced Network Classes (NCs), leading to a higher share of traffic for the probes in N.America/Europe. Interestingly, NCs with the most number of pageloads, although showing good improvements (11-13% at median), are not the ones where we observe the highest benefits, owing to their good bandwidth, low RTT connections. We observe that the less-dense NCs still outperform Default (by more than 8% at median), since Configanator’s exploitation arm is able to generalize to a modest extent by using data collected across all NCs.

5.8 Discussion and Limitations

Security and Equilibrium: Potential implications of self-learning systems include adversarial attacks [475] or oscillations. We are working to formulate the interactions between different instances of Configanator (i.e., deployments by different CDNs) as a game-theoretic problem to understand our system at equilibrium.

Management Overheads: Dynamically reconfiguring the CDN’s protocol stack complicates performance diagnosis. We plan to investigate methods for reducing this complexity, e.g., minimizing the number of active configuration combinations. Further, different configurations may vary in their resource-consumption at the CDN edge and we plan to investigate the configuration associated resource-overheads in the future.

Data Bias: Configanator’s data-driven workflow can be impacted by the inherent biases of trace-driven systems [61], e.g., choice of configuration can have an impact on the feedback loop’s decision features. We leave a more comprehensive analysis of biasness to future work.

Testbed Limitations: Owing the lack of cellular connections and devices in the testbed, our simulator is unable to emulate different end-user devices and cellular last-miles. Although the dataset from GlobalCDN covers diverse last-mile connections and devices, we plan to explore systematic approaches to incorporate this diversity in the testbed.

Trace Limitations: While several of our traces capture end-to-end behavior (GlobalCDN, Pantheon, FCC), two of our traces do not. Specifically, CAIDA and MAWI traces are from core router and we recreate end-to-end behavior by matching data with ACKs: this recreation can introduce some imprecision into our latency, loss and BW calculations.

NC Size Bias: As demonstrated in the evaluation, connection homogeneity within an NC (due to small NC size) favors Configanator. This bias is prevalent in two of our traces, FCC and Pantheon (comprising synthetically generated flows). However, this does not hold for the realistic traces (e.g., GlobalCDN) which are mainly focused in the evaluation when discussing the size bias, and still shows improvements over the Default.
<table>
<thead>
<tr>
<th>Knob</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TargetAlgo</td>
<td>Sets corresponding tuning algorithm.</td>
</tr>
<tr>
<td>TargetNC</td>
<td>Controls the clustering strategy for NCs.</td>
</tr>
<tr>
<td>init_samples</td>
<td>Number of samples to initialize an NC.</td>
</tr>
<tr>
<td>NCSpread</td>
<td>Controls the performance spread that bounds a NC, and hence the number of cluster (K discussed in § 5.3.2).</td>
</tr>
<tr>
<td>AllowedConfig</td>
<td>Limits the space to disallow certain configurations.</td>
</tr>
<tr>
<td>PerfMemory</td>
<td>Length of history for configuration’s performance over time.</td>
</tr>
<tr>
<td>UpdateLatency</td>
<td>Set latency b/w central Config. Manager and servers.</td>
</tr>
<tr>
<td>UpdateFreq</td>
<td>Controls the time after which a model is updated.</td>
</tr>
<tr>
<td>ChunkSize</td>
<td>Controls the time window for goodput, RTT and loss-rate measurements from packet traces.</td>
</tr>
</tbody>
</table>

Table 5.4: Simulator knobs

5.9 Conclusion

In this paper, we argue that “one-size-fits-all” approach to configuring web server’s network stack results in sub-par performance for end-users, especially those in emerging regions. Due to the ever-expanding nature of Internet, all end-users do not face similar network conditions. This argument stands in stark contrast to the traditional setup of today’s web serving stacks where a single configuration is used for a divergent set of users.

This paper takes the first step towards realizing heterogeneity and fine-grained reconfiguration in a principled and systematic manner: our system, Configanator, introduces a principled framework for learning better configurations, than the default, for a connection by systematically exploring the performance of different configurations across a set of similar connections. We demonstrate the benefits of Configanator using both a live deployment and a large scale simulation.

5.10 Supplementary Results

In this section, we present the supplementary discussion and evaluation results.

5.10.1 Fingerprinting Configurations

Our fingerprinting techniques are inspired from recent works [208, 364, 432, 544]. Our tool inter-operates with TLS and infers configurations in the following ways: (i) HTTP configurations are visible to client during the connection setup and are fingerprinted from the server response, (ii) TCP configurations like RWIN are scraped from the packet headers, (iii) TCP initRTO is measured by emulating a loss during TCP handshake (i.e., by not acknowledging SYN packet back to the server), and measuring the time it takes the server to retransmit the SYN/ACKs, (iv) For TCP ICW, a big enough object URL is scraped from a website, the corresponding object is fetched and the number of packets sent by the server in first RTT is measured. Further, we use MSS=64B to trigger higher number of packets from server. We used AWS in respective regions as the vantage points for fingerprinting the configurations.
5.10.2 TLS Fingerprinting for Device Identification

Recall that instead of the traditional User-Agent string, Configanator uses TLS fingerprinting for device identification as it allows device inference in early stages of the connection (prior to the HTTP version negotiation through ALPN). To evaluate its efficacy, we leverage a dataset from GlobalCDN, comprising 3.6M requests. The dataset consists of server logs and captures User-Agent strings from HTTP GET requests and the TLS fingerprint of the respective connections. The dataset includes 14.5K unique User-Agent strings and 3.2K unique TLS fingerprints.

Figure 5.19 plots the number of unique User-Agents (UA) that map to a TLS fingerprint. Ideally, a single UA should map to a fingerprint, thereby accurately identifying the corresponding device. However in practice, we observe that the one-to-one mapping is limited only to 34% of the fingerprints, with the rest mapping to at least 2 UA. We observe that complementing the TLS fingerprint with the end-user IP-prefix helps in improving the accuracy, with 78% of the IP/24 and TLS fingerprint mapping to a single UA and 96% mapping to at most 8 unique UA. We observe that for the cases where a single fingerprint maps to multiple UA strings, there are only minor differences, e.g., different browser versions, difference in OS’s minor version (Android 6.0 vs 6.1.1).

In Figure 5.20, we further compare the two device identification techniques for clustering similar connections together. Using a dataset of 89K PLT measurements from GlobalCDN, we run our Network Class clustering using either User-Agent or TLS fingerprint as the basis for device identification. We compute the Euclidean distance of each connection PLT from its cluster’s center and the figure plots the ratio of the distance. We observe that the ratio is between 0.98 and 1.00 for the overwhelming majority of the pageloads, indicating that the two techniques perform fairly similar. Hence, device identification through TLS fingerprinting provides nearly similar accuracy to the User-Agent strings, with the added benefit that the device is identified prior to negotiating the HTTP version, whereas User-Agent string can only be inferred through HTTP requests headers (received after HTTP version negotiation).
5.10.3 Passively Recording Network Conditions

`Configanator` passively collects goodput and packet loss rates for the IP-prefix (/24) and builds a historical archive (§ 5.3.2). When an IP connects, `Configanator` uses the handshake RTT and looks-up the goodput and packet loss rates from the recent session for the IP-prefix (/24) to aid in classifying the user into her Network Class. `Configanator` prototype uses Apache logs to collect information about user IP, the requested content, content size and download time. Additionally, per-connection TCP statistics are captured through Apache TCP Info plugin [450]. Using this information, `Configanator` calculates bandwidth (goodput) and packet loss rates on a per IP-prefix (/24) basis. Although we use heuristics for stable goodput and loss calculations, e.g., ignoring small objects, the goodput estimate may still under-estimate actual network bottleneck due to TCP mechanics (e.g., slow start phase). Consequently, the use of such measurements in the testbed (§ 5.6.1) may emulate lower bandwidths and higher loss rates (emulated loss plus induced buffer overflows) than the actual bottleneck links.

5.10.4 Gaussian Process and Network Class Discussion

**Bootstrapping GP:** The first step of learning is to acquire data to bootstrap the Gaussian process. The bootstrap methodology is crucial for ensuring that the Gaussian-Bandit quickly finds good direction to explore. Recent works [29, 151, 71] have demonstrated the applicability of three distinct bootstrapping approaches: (i) *random*, in which the initial configurations are randomly selected; (ii) *domain-specific*, in which prior domain knowledge, captured through operator interviews or offline simulations, are used to rank configurations to sample; (iii) *Latin Hypercube Sampling* (LHS) which divides the input space into partitions and selects a sample from each partition to spread the samples evenly across space [474]. In this work, we use LHS to bootstrap the learning process. LHS has been found to aid bootstrapping Bayesian optimization by reaching an optimal decision quicker [333]. We observed LHS to speed up exploration in comparison with others by reducing the number of optimization steps by 2-3X, as the bootstrapping samples are spread evenly across space. A perfect rankings of configurations cannot be known prior to actually testing configurations, leading to ranking-based bootstrapping being sub-optimal to LHS.

**Individual GP models for each website/Network Class:** Bayesian Optimization is traditionally used for mapping configurations to their performance per workload (e.g., cloud configuration to cost [29]). Due to network dynamics and their implications on web performance [255, 520], a separate BO/GP model is required to map configuration performance for each workload (network condition and website), leading to individual exploration for each workload. The lack of cross network/website exploitation (due to separate BO models) makes a solely BO-based technique unfit for `Configanator`. Intuitively, the system should be able to generalize across networks and can use the already learnt pattern from other networks to a new network, e.g., HTTP/1.1 is optimal at high RTT, high loss for a complex website, no matter the bandwidth [520].

Figure 5.23 presents the GP model for two websites for the same set of configurations (x-axis) and the same Network Class (NC). In Figure 5.21, while GP has estimated the curve for `youtube` with high confidence, `amazon` requires more data samples (wide confidence interval for configuration 7, 8 and 9). The different configuration-performance curves require a separate GP model for each website/NC for correct modeling of
a configuration’s performance and effective exploration, as the configuration-performance curve is distinct for every website and Network Class.

**Figure 5.23:** GP config-performance curve.

**Impact of performance spread within NC:** Next, in Figure 5.24, we leverage the $\text{NCSpread}$ knob in simulator (Table 5.4) to test different bounds for Network Classes (NC) clustering. Recall that $\text{NCSpread}$ controls the “K” for Kmeans clustering by selecting the lowest K that bounds the standard deviation of PLTs for a cluster’s constituents within a specified threshold ($\{1, 2.5, 5, 10\} \%$ in the Figure 5.24). Determining the right K involves iterating through K values and is a three step process: (i) NC features – network characteristics (bandwidth, latency, loss rate), AS information (ASN, geo-location), and device type – from past connections are clustered using a given K, (ii) For each cluster, the list of PLTs observed for its members connections is generated and is normalized by the median PLT of the list$^{11}$, (iii) The standard deviation for each list is computed and, based on how far is it from the median and the $\text{NCSpread}$ limit, the decision to converge on the given K or test a different K is made.

Figure 5.24 uses the testbed generated data from § 5.6 and plots the error in GP’s estimate for five randomly selected configurations. The error is calculated as the absolute difference of GP’s PLT estimate for a

---

$^{11}$As there might be multiple websites, there is one list per website. Further only PLTs for default configuration are used.
configuration and the actual PLT, and the boxes plot the error distribution observed for the various clusters (corresponding to the NCSpread value) and the websites. Note that, a small error is always expected due to the inherent variability with PLT measurements. The 5% limit NCSpread performs fairly close to the lower bounds, while also requiring a lower K: 7% lower K value than the 1% NCSpread threshold. This analysis serves as the motivation for using 5% value in the simulator.

Figures 5.21 and 5.22 further visualizes the confidence intervals for the GP models for 2 websites. For the high spread case (10% NCSpread), connections from slightly different networks are mapped to the name GP, resulting in wider confidence intervals, and leads to inefficiency with acquisition function’s next configuration suggestion.

5.10.5 Deployment Considerations

Data-driven systems [256, 25, 255, 197, 254] traditionally use a split-plane architecture where a modeling layer (responsible for ingesting huge amounts of data and updating models) runs at a slower granularity than the decision layer (responsible for applying modeled decisions for users at real-time). Configanator’s architecture uses a split-plane model which leverages the different computational requirements of Configanator’s workflow: As demonstrated in Figure 5.4, in the slow path, the Configuration Manager collects telemetry from the web servers, uses this telemetry to update the learning model, and installs the configuration rules created by the model into the web servers. In the foreground, each web server uses the pre-installed rules to apply configuration to each connection and periodically collects telemetry from each connection.

The first phase, the background process, is time-consuming because of the process of updating the learning algorithms and Network Classes. The second phase, a fast, real-time process that applies the configuration rules to each inbound user connection, is run at the edge on each web server and provides low-latency, dynamic tuning. We note that although this decoupling results in the fast-path using stale information, we observe that this stale information still provides near-optimal performance [197].

5.10.6 Supplementary Evaluation Material

Figure 5.25: Simulating pageload for a client
Evaluation Setup

Simulation workflow: Figure 5.25 presents the workflow for simulating pageload performance. The client sessions are extracted from the real-world datasets and are modeled as time-series. Since we use 5s for measuring the network characteristics from the trace (a tunable knob as discussed in § 5.6.1), each linear state for BW, RTT, Loss in Figure 5.25 is at least 5s long. We extract an IP distribution from the trace to model the temporal aspects of client’s connections (time at which a client connection (or IP) is seen in trace), i.e., the user sessions are fed to the simulator in the order they are observed in the real-world trace.

The simulator takes the goodput, RTT, loss rates at a certain time from the session time-series, client info (IP, ASN) and the target website to load as input. Using these features, it consults the configuration to test from the learning framework. Once the target configuration is known, it leverages the PLT-Tensor to map the network characteristics \{goodput, RTT, loss-rate\}, website and configuration to the eventual PLT. Note that, we assume that the network characteristics stay stable throughout the lifetime on a single pageload, supported by recent studies that TCP connection is piece-wise stationary and each segment stays stable in the order of tens of seconds to minutes [57].

Table 5.4 further summarizes a number of simulator knobs that allow us to emulate and test a variety of scenarios.

Dataset description and breakdown: While the GlobalCDN, MAWI and CAIDA datasets are adequately described in § 5.6.1, here we provide details for the other two datasets.

The Pantheon dataset [542, 543] comprises of synthetically generated TCP flows across the different parts of the world. We collected three month’s worth of data (May to July 2018) from Pantheon’s website [542]. For the generated flows, the dataset logs the flow IDs, packet ingress/egress timestamps, packet sizes and one-way delay. Using these fields, we calculate the goodput, RTT and loss rates between each pair of end-point and, similar to the case for GlobalCDN, MAWI and CAIDA datasets, generate the time-series for the network characteristics. These end-points (vantage points) range from AWS deployments to university networks and cover multiple last-mile connection types. The FCC dataset is collected by the Measuring Broadband America program [181] and consists of a nation-wide study of end-user’s broadband performance and an accompanying dataset. This dataset provides coarse granularity measurements in form of bandwidth, latency and loss rates distributions measured for real-world users. We use these distributions to generate synthetic traces, similar to [25].

The breakdown of ~21.4M sessions is as follows: 8.2M from GlobalCDN, 2.7M from MAWI, 8.1M from CAIDA, 1.6M from FCC, 800K from Pantheon. The cross-regional nature of our datasets provide coverage over a wide range of representative network conditions, e.g., while FCC and CAIDA cover connections in U.S., MAWI dataset is from East Asia. Further, GlobalCDN and Pantheon [543] are even more diverse with connections from countries across the globe.

SpeedChecker and vantage points: SpeedChecker [319] is a platform for global Internet measurements, with vantage points deployed in over 170 countries and thousands of ISPs. SpeedChecker provides an API to conduct automated measurements ranging from ping, DNS, web pageloads to video tests. We leveraged vantage points (windows machines) on this platform for conducting the pageloads. The API call requires CountryCode and Destinations (a list of URLs to load). Vantage points (probes) from the specified country
Figures 5.26 presents various distributions about our vantage points. 5.26 compiles the distribution of network conditions observed for each pageload. The vantage points vary across the three dimensions and have mostly RTTs greater than 100ms. 5.27 presents the number of pageloads per prefix. We observe a heavy tail distribution, where certain vantage points conducted more pageloads than others, e.g., Europe, N.America had 4X more pageloads than Asia and Africa due to the higher number of the SpeedChecker clients in the developed regions. Africa had the smallest number of vantage points among all continents and the hourly limits were frequently reached, resulting in a lower number of total pageloads. Note that, diverse network conditions were still observed for the vantage points (Figure 5.26) in spite of this skew in vantage point location. We further observe that 90% of the vantage points have unique IP-prefix (/24), showing that they are distributed and are not placed in a single facility, in the same subnet.

**Configanator Performance for CAIDA, FCC and Pantheon Traces**

Figure 5.29 presents the distributions for Configanator’s PLT improvement for the CAIDA, FCC and Pantheon traces. These figures complement the results in § 5.6.2 where we could not add the results for all the traces due to space limitations. CAIDA and FCC traces are collected from U.S.A and mostly cover high
bandwidth, low RTT/loss connections, e.g., p95 RTT is 60ms. Following the trend observed in Figure 5.8, we observe their PLT improvements over default to be lower as compared to other traces. Especially FCC dataset covers broadband connections and we observe the lowest p95 PLT improvement for FCC among all the datasets. Nevertheless, the improvements are still substantial with 610-640ms decrease in p95 PLT. On the other hand, Pantheon traces cover wider range of networks, often across continents, and result in up to 850ms improvement at tail.

**Bandit Contribution**

Figure 5.30 uses the same convergence analysis as Figure 5.10 and plots the percentage of connections that uses a certain bandit. Initially GP bandit is largely used for a guided exploration. However, as more data is collected, DT bandit starts to overshadow the GP bandit, highlighting that a per-NC guided exploration is over-shadowed by cross-NC exploitation, when large data is available.

**Bandit Performance at Tail**

Figure 5.31 focuses on tail by dividing the entire trace into one minute segments and plotting the distance to optimal for the worst-case tail of each minute. *Configanator’s* use of bandits enables it to perform better than individual bandits, being closer to optimal by more than 7%.
Configuration Stability

Figure 5.32 plots the number of connections across NCs after which the DT-bandit’s decision stays stable, i.e., configuration decision for the NC does not change. While for the median NC, the configuration choice becomes stable at ~400th connection; we observe that it can take as much as 10K connections to reach the final configuration for some NCs. We observe the DT-bandit to stuck on a near-optimal configuration for these NCs. Down the line, the epsilon-bandit, randomly exploring, finds the optimal configuration and updates the NC. We note that Configanator switches to DT-bandit in the first 10-15 iterations for these NC, highlighting that the GP model’s EI threshold was reached very early, and the initial exploration through GP was not very beneficial in uncovering the optimal configuration.

Design Choices for Network Classes

We use GlobalCDN dataset to evaluate design choices for classifying similar users together. We compare Configanator’s clustering with: (i) IP-Prefix clusters /24 users together, (ii) Hobbit [297] improves /24 groups by merging dis-contiguous /24s based on co-location in Internet topology and homogeneous performance, (iii) Latency Driven inspired from AP-Atoms [408] where users with similar latency are grouped together, (iv) since CDNs group users based on their performance similarity [462, 101], CDN Aggregation use the natural CDN grouping and assigns all users mapped to a PoP to the same NC. We extract these mappings from the GlobalCDN dataset and, as these mappings can vary over time, build a time-series of user to CDN PoP mapping.

![Figure 5.33: Impact of clustering on # of NC](image)

![Figure 5.34: Spread of TTLBs within NC](image)

Figure 5.33 and 5.34 plots the number of NCs and the spread of TTLBs within an NC for the different strategies. Ideally, Configanator favors small number of NCs and aims for small to negligible variations within NC performance metric, as the goal is to cluster similarly performing users together (§ 5.3.2). We observe Hobbit subnets /24 groups to have a poor coverage over the trace (Hobbit only covers 12% of prefixes in GlobalCDN dataset), with non-Hobbit /24s being treated as individual groups, leading to similar results as IP-Prefix (figure 5.33). Although NCs built by Hobbit and IP-Prefix have lowest performance divergence, (low std. dev. in figure 5.34); Configanator NCs are almost similarly compact, while using less than half number of NCs. Although Latency Driven uses least number of NCs, the lack of device, bandwidth, loss etc.
information leads to diverse users being grouped together (high TTLB std. dev.). Similarly, since CDNs maps user based on latency to their closest PoPs, network and device heterogeneity still exist (e.g., the closest PoP to a user can be 10ms-600ms [101]), leading to highest performance variation within an NC for CDN Aggr.

We further modified the simulator to explore Configanator performance when different NC techniques are used. We observe that Configanator out-performs the rest for the majority of the pageloads. The prefix and CDN based approaches either do not account for network dynamics or overfit to specific regions respectively. Latency driven performs slightly better but ignoring the important metrics, like packet loss and bandwidth, degrades its effectiveness.

**PLT Variability**

PLT measurements are inherently noisy [378] and the variability in PLT can disrupt the learning algorithm’s model, e.g., GP is sensitive to noise [310]. Using data from a web performance observability company (NewRelic [271]), we modeled PLT variability distribution and used it to introduce variability in testbed-generated PLT-Tensor. Figure 5.35 plots a PDF of the variations with x-axis as the mode PLT multiple (x-axis is PLT normalized by mode PLT). We fit an Erlang curve to the observed PDF, owing to its right-skewed, long-tail nature. Using PLT from PLT-Tensor as the mode PLT (since mode PLT is the most stable PLT measurement), the PDF is used to calculate the noisy PLT observed by a real-world user.

Figure 5.36 plots the extent to which Configanator decisions (in face of PLT noise) are optimal, compared to the case when there is no noise (Configanator-NoVari). A proximity score of 1 indicates that Configanator decisions stay exactly the same for both (noise, no-noise) cases. Leveraging the PerfMemory knob, we test 2 scenarios with different length of historical memory. Configanator uses this historical memory to amortize the impact of any sudden change in performance metrics. We observe Configanator’s decisions to slightly deteriorate in face of noise. However the extent is mild at worst — with the system still assigning the optimal decisions more than 95% at median.
Dissecting PLT Degradation

As shown in Figure 5.9, all algorithms result in some PLT degradation. Figure 5.38 plots the percentage of sessions that faced PLT degradation and further divide them into the root-causes. Our observations are as follows: (i) During exploration, multiple configurations are tested and may result in degradation. Around 5-6.2% of the sessions in two of the datasets are such exploration steps. (ii) As network conditions change over time, Configurator’s estimate of historical network characteristics for an IP-prefix may diverge from the actual network. The stale information is used for classifying the connection into an NC and predicting the optimal configuration. Due to the global nature of GlobalCDN dataset, we observe a higher network churn, with 6.6% of total sessions resulting in PLT degradation due to stale NC. Only a small proportion of sessions in MAWI dataset (∼0.1%) resulted in PLT degradation with correct NC view, indicating that the exploitation arm momentary got stuck at a sub-optimal configuration.

CM Design Choices

Configuration Manager Design: CM can run locally in a PoP or centrally within a data-center [197], trading-off between data-size to learn and the speed to react to changes. We evaluate both scenarios in our simulator: In the local design, there’s a separate CM for each trace, while for the global case there’s a single CM for all traces. To simulate each scenario, we vary the latencies between CM and the web servers. We observe that while the global CM is able to make slightly better predictions at the tail (2% better than local), the difference at median is negligible. Despite the larger data set, global CM is not significantly better due to distinctly diverse network conditions across regions (only 17% NCs are common in U.S and Japan traces).

Frequency of model updates: Next, we analyze the impact of updating our performance model less frequently: we explore a range of values from every 2 minutes up to every day. We observed performance to stay relatively stable at the median, whereas hourly or lower update intervals result in ∼8% better improvement at tail, than a per-day granularity.

\[\text{It takes } \sim 2 \text{ minutes to update the models for 10K sessions.}\]
Flows Through Access Link

We use packet trace from [446] to measure the typical number of TCP flows through an access link. Figure 5.39 presents the number of TCP flows with at least 10Kb data transferred, in a 60s time interval. On the median 60s time interval, we observe around 25-30 flows competing through the access link.

Additional Micro-benchmarks

In addition to the system benchmarks, we also evaluated two alternate design choices: VMs and LD_Preload. For VMs, we used one VM for each configuration and used Open vSwitch (OVS) [190] for routing flows to the appropriately configured VM. We explored the use of LD_Preload to intercept system call and tuned socket using setsockopt(). In comparing both choices with Configanator, we observed that the VM-based approach introduced a 20% increase in latency where as the LD_Preload introduced a much smaller latency of 2.2%. We also observed overheads for CPU and Memory utilization: the VM-based approach introduced 30% (memory taken by the guest OS) while LD_Preload introduced a 5% increase.
Chapter 6

JS Capsules: Characterizing JavaScript’s Memory Overhead for the Mobile Web.

6.1 Introduction

Mobile phones have become the primary gateway to the web, with the number of mobile subscriptions surpassing the total human population by 1.3 Billion [524] and contributing over 55.8% of website traffic worldwide in 2022 [110]. While mobile apps are a key driver behind the high mobile traffic share, studies show that mobile browsers also serve as an important gateway to the web [387] and have a direct impact on retailers’ revenue [261]. Despite the importance, mobile web is still plagued with performance issues [87, 372], primarily due to the friction between the high resource demands of the complex websites [86, 12] and the modest hardware of mobile phones [518, 168, 12]. Consequently, and due to the tight coupling of QoE with revenue [145, 78, 82, 396, 230], understanding the device limitations in mobile-web has recently garnered significant attention.

Recent studies from academia have tackled the resource limitations in mobile devices from different axes, including network [167, 379, 280], computation [518, 372, 123], energy [94, 126, 489], JavaScript inefficiencies [99, 168, 280, 98, 290] and, to some extent, web memory overhead [123, 126, 416, 269]. Specifically for the web memory overhead, Chrome, V8, and web development teams from multiple organizations have proposed several recommendations and optimizations, such as optimized JavaScript parsing, compilation, and execution [413, 394, 112, 486, 162], object compression [503, 411], heap memory leaks [62, 388, 273, 272], and lazy loading/rendering [17, 418, 68, 177, 178]. Further, others have proposed “signals” to detect memory issues, such as device hardware info [515] and real-time heap memory inference [369, 17], in order to provide developers with toolkits to detect and improve the memory performance of their applications.
While these recommendations have been instrumental in shaping the state of the web, today, we lack techniques that can enable a developer to holistically quantify and qualitatively understand the memory overhead of their website code, specifically JavaScript, a cornerstone of today’s dynamic and interactive web ecosystem [398, 99, 168, 280, 98]. Besides providing a deeper understanding of JavaScript memory efficiency, such techniques can provide the developers with a systematic way for selecting the appropriate optimizations to fix their website-specific memory problems or, if none exist, revise and engineer new solutions to fit their domain-specific needs. In fact, Chrome teams have written multiple guides to identify and fix memory problems [62, 272, 137, 345]. However, in our experience, these recommendations either lack the coverage or fine-granularity required to tackle the challenges with JavaScript memory accounting, as we discuss next.

Investigating JavaScript memory overhead requires holistic measurement techniques to capture fine-grained allocations (i.e., how much memory is allocated?) and identify their source (i.e., what and why do the JS functions incur memory overhead?). The two properties tackle the challenge of understanding JavaScript memory from two different angles: while the former enables a developer to accurately measure function-wise memory and identify the problematic functions, the latter provides visibility into the nature of JavaScript and their impact on browser mechanics (i.e., browser events), essential for understanding the source of allocations. However, realizing both of these tasks in practice for an arbitrary website comes with a significant amount of challenges.

• First, a JavaScript function often leads to a wide range of browser events that span across multiple browser components, from JavaScript engine V8 [502], networking components, rendering engine [215] to the GPU [494]. Especially, JavaScript’s dynamic interactions with a webpage’s Document Object Model (DOM) often cascade into events that require memory from different sources, e.g., V8 memory for parsing/compiling the code, building objects (such as strings), compositor memory for storing and updating layout information, and GPU memory for generating animations, etc. Thus, JavaScript incurs a direct and an indirect memory overhead, and a fine-grained, cross-component view is required to correctly attribute the memory overhead to the respective JavaScript functions and their cascading events.

• Second, identifying the key root-cause of memory overhead, i.e., the specific browser mechanics and their properties involved in parsing, compiling and executing a function, is complicated due to the fine-granularity of events and the wild-wild-west nature of JavaScript. For the median website, 84% of the browser events during a pageload are precipitated in response to a JavaScript function and are called at the granularity of tens of milliseconds. Further, websites comprise thousands of JavaScript functions (≈2800 at median for Alexa top 1K) and developers have the freedom to use a vast range of arbitrary APIs and external libraries. While the fine-granularity of events makes it challenging to attribute browser events to a specific function, the arbitrary nature of functions makes it challenging to draw generalizable insights, thereby requiring analysis techniques to extract the key features that impact memory.

Unfortunately, the existing recommendations solely rely on OS-level [154, 405] or Chrome-based [137, 491, 11] memory analysis tools that either provide incomplete coverage or coarse-grained visibility. For
instance, while HeapProfiler [11], recommended in [62, 272], measures memory on a functional-basis, it only focuses on the JavaScript heap. Consequently, its scope is limited to the V8-component, thus providing incomplete coverage as our measurements show that V8 only contributes \( \sim 27\% \) of the JavaScript memory at the median. On the other hand, while MemoryInfra [139], recommended in [137, 345], measures memory across all the components, it does not measure JavaScript memory in isolation. Particularly, it measures the complete overhead incurred during the page load and does not tie back the cross-component allocations to fine-granular JavaScript functions, thereby, making it challenging to infer if allocations across the V8 boundary are an artifact of JavaScript indirect memory overhead.

To tackle these challenges, we propose a novel measurement framework for accurately capturing fine grained measurements of JavaScript functions’ memory overhead. The key insight behind our design is that most browsers execute JS in a single-threaded fashion [518, 330, 376] per frame and use a single main thread for scheduling JS events [330] to guarantee safe concurrency for the shared DOM data structure. Leveraging this property, we decompose a website’s JS into isolated blocks called JS-Capsules, formally defined as a set of one or more JS functions, represented by a unique, non-overlapping time interval (i.e., start and end timestamps). Leveraging the timestamps and the serial-access model, we capture a JS-Capsule’s cascading events by decomposing the page load process along a browser’s thread boundaries and use low-level Chromium APIs [134, 135] to measure fine-grained memory allocations. In essence, JS-Capsules provide us with a framework for fine-grained visibility into the JavaScript functions and the resulting cascading events, as well as, a complete coverage over the various sources of memory allocations.

While the fine-grained data generated by the JS-Capsules provide visibility into the deeper JS and browser mechanics, extracting generalizable insights for arbitrary JS functions/libraries and diverse websites requires specialized analysis techniques to extract meaning out of the fine-grained allocations. For this purpose, our framework uses clustering, statistical, and machine learning techniques to identify and understand the JavaScript memory allocation patterns. At a high-level, the goal of the clustering is to create an abstraction class where developers can reason about optimizations that can be applied to JS-Capsules with different natures of allocations and easily understand fundamental browser interactions behind the overhead.

The design of JS-Capsules is based on Chromium architecture, the underlying framework that powers popular browsers like Chrome, Brave, Edge, Opera, etc. [108]. In 2022, Chromium accounted for over 67.6% of the global mobile browser share [472]. Given Chromium’s broad use, we believe that by building our design on its architecture, we ensure that our insights and observations are broadly applicable across popular browsers. This work makes the following contributions:

- We propose a novel measurement technique to dissect JavaScript’s direct and indirect memory overhead in a fine-grained and holistic manner. Leveraging our framework, developers can diagnose memory efficiency of their website’s JavaScript functions and automatically localize the overhead to the problematic functions.

- We propose and develop a root-cause methodology for attributing the JavaScript’s memory overhead to cascading browser events that span across various components, ranging from V8 to the rendering engine, GPU, etc. Our root-cause methodology enables developers in gaining visibility into the key
browser mechanics and JavaScript/website properties that have a direct impact on the memory overhead.

• Leveraging this toolkit, we present the first-ever characterization of JavaScript memory for the Alexa top 1K websites. While JavaScript memory overhead is generally discussed in V8’s context, we show that the bulk of JavaScript memory overhead (73% for the median website) originates outside of V8’s boundary and further show sensitivity of JavaScript memory overhead to a diverse range of features such as V8’s parsing behavior, website’s DOM properties and dynamic interactions between JavaScript and DOM, among others. A key significance of our measurements results is that it unveils the potential optimization targets, if one aims to optimize JavaScript memory beyond V8’s context.

• We present several case-studies where we leverage our tool to diagnose memory for different types of websites and further dissect the overhead of advertisement and tracker JavaScript, as well as popular JavaScript libraries and frameworks.

• We evaluate several cutting-edge JavaScript, browser and web optimizations, geared toward improving JavaScript efficiency and leverage our methodology to dissect their potential benefits and limitations.

Our measurements show that JavaScript memory overhead is an artifact of various design choices taken by the developer. While some of these factors are intuitive, such as code size, extent of reliance on client-side JavaScript to build websites, etc., others are non-intuitive, such as DOM design, layout complexity, usage of specific browser components or JavaScript libraries, and require a rethinking of website design to reduce resource wastage. Further, our measurements show a significant website-specificity of JavaScript memory overhead and unveils the potential optimization targets for different types of websites. We further show that most over-the-counter optimizations lead to modest memory savings and the optimizations that have a significant impact require either developer effort to introduce extensive changes to the website design, or significant changes to the end-to-end web infrastructure.

The paper is structured as follows: Section 6.2 presents the background on Android OS memory management, Chromium architecture and the page load process, and Section 6.3 presents the motivation for JS memory analysis and discusses the challenges associated with capturing fine-grained memory analysis using existing tools. Section 6.4 presents the design and implementation of our framework (JS-Capsules) and Section 6.5 discusses various analysis techniques we leverage for the memory analysis and attribution. Section 6.6 validates our methodology, while Section 6.7 presents the measurement results for Alexa top 1K websites. Finally, Section 6.8 evaluates several optimizations.

6.2 Background

In this section, we discuss Android OS’s memory management, the architectural details of the Chromium browser and provide a high-level overview of the page load process, with a specific focus on JavaScript. We also discuss the performance tracing and the role of different memory allocators for loading a webpage.
6.2.1 Android Memory Management

Android runs on the premise that free memory is “wasted” memory and tries to use all of the available memory [128] at all times [130]. Unlike traditional Linux, Android does not use virtual memory techniques (e.g., swapping) to ease memory contention and instead relies on Garbage Collection (GC) [131] and Out-of-Memory (OOM) killer [277]. Garbage collection is a mechanism for reclaiming unused memory and, at a high-level, has two goals: (i) find memory references that cannot be accessed in future, and (ii) reclaim such memory objects [131]. Garbage collection often incurs CPU overhead and may even stall the application [148]. In the extreme cases when enough memory cannot be reclaimed, Android uses OOM killer to identify and terminate one or more processes (based on pre-configured priorities), thereby reclaiming memory space for a higher-priority process.

6.2.2 Chromium Architecture

Chromium is the underlying framework that powers popular browsers like Chrome, Brave, Edge, Opera, etc. [108]. In 2022, Chromium accounted for over 67.6% of the global mobile browser share [472]. Given Chromium’s broad use, by building our design on its architecture, we ensure that our insights and observations are broadly applicable across popular browsers. On Android, Chromium uses a multi-process architecture with three types of processes — Browser, GPU and Renderer — which are responsible for different aspects of loading a webpage [283, 107].

- The Browser process is in charge of UI, disk, and network I/O and, in particular, controls the address bar, browser UI buttons, file access and makes network requests.
- The GPU process interfaces with the GPU device for GPU tasks. The architecture also specifies an in-process GPU implementation [494] and selectively offloads tasks to the device GPU.
- The Renderer process “controls anything inside of the tab where a website is displayed” [283] and performs the core functionality of parsing and rendering webpages. It includes JS compilation engine V8 [502, 170], parses webpage code (e.g., HTML, CSS, JavaScript) and prints the webpage on screen. While the other two processes are shared across different tabs, a new Renderer is spawned for each new tab. Furthermore, for security and isolation purposes, the browser spawns multiple Renderers if the webpage contains cross-origin content (typically in iframes) [284, 335].

6.2.3 Page Load Workflow

The page load process begins with a user-initiated request which downloads the root HTML file. The HTML file specifies tags for different content types (e.g., a or p for text, img for images, script for JavaScript) and a hierarchy, i.e., sibling or parent-child relationships between the different tags. The HTML file may also include references to other web objects (e.g., URLs for images, JS/CSS files).

As soon as the first chunk of HTML file is downloaded, the Renderer initiates HTML parsing to construct the Document Object Model (DOM) [336], an object-oriented representation of a webpage’s structure and
content. The DOM comprises nodes for the different tags and is constructed as a tree to maintain the hierarchy between the nodes. The Renderer evaluates the styles for the DOM nodes (typically defined through style attributes in HTML or CSS) to yield a render tree, which serves as the basis for determining the layout of nodes (i.e., position and size) and painting the visible nodes on screen. While parsing the HTML, the referenced objects are downloaded\(^1\) and the Renderer progressively paints the webpage.

Among the referenced objects is JavaScript (JS for short), the key focus of this paper. At its core, JS provides a programmatic interface for dynamically interacting with the DOM. Developers use JS to dynamically fetch objects, add new nodes, modify existing nodes (e.g., position, size), modify style rules (e.g., CSSOM [115]) or perform any required computations (e.g., maintaining sessions or state). To avoid any conflicts in building or modifying the DOM tree [518], the browser blocks HTML parsing while any embedded JS is evaluated.

JS is a single-threaded language [200]. Moreover, Chrome instantiates a separate instance of the V8 engine within each Renderer which, in turn, maintains a single memory heap (used to allocate memory for the JS program) and call-stack [525]. Browsers execute JS in a single-threaded fashion [518, 330, 376] per frame and use a single main thread for scheduling JS and most rendering events [330]. This serial-access model simplifies page development and guarantees concurrency for the shared DOM data structure.

### 6.2.4 Chromium Performance Logging

Chromium provides developers with several diagnostic tools [421, 497] to understand the performance of page load. For the target “trace categories” [83, 498] defined by a developer, the comprehensive logging infrastructure captures a variety of browser activities, such as networking, CPU, GPU, V8, rendering tasks, etc. The trace, formatted as a JSON, comprise a sequence of events where “cat” and “name” fields define the category (e.g., devtools.timeline [491], v8.cpu_profiler [415], etc.) and name of the event (e.g., v8.compile, FunctionCall, Layout, UpdateLayoutTree, InvalidateLayout, Paint, etc.), respectively. The logs also contain information about the browser architecture, such as process metadata that maps process-IDs to their type, i.e., Browser, GPU or main/iframe Renderer process.

Two sets of metadata are logged for each event [467]: while general metadata, including start timestamp, duration, process/thread IDs and phase (marking the nature of the event as sampling, duration, instant,\(^1\)The downloaded objects are only evaluated when needed [518].

<table>
<thead>
<tr>
<th>Event class</th>
<th>Event name</th>
<th>Event-specific metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading</td>
<td>ParseHTML</td>
<td>url, startLine, endLine, line, frame</td>
</tr>
<tr>
<td>Scripting</td>
<td>EvaluateScript</td>
<td>url, lineNumber, line, start, end, frame</td>
</tr>
<tr>
<td>Scripting</td>
<td>v8.compile</td>
<td>url, fileName, lineNumber, column, line, stream</td>
</tr>
<tr>
<td>Scripting</td>
<td>FunctionCall</td>
<td>url, functionName, scriptId, lineNumber, column, line, frame</td>
</tr>
<tr>
<td>Rendering</td>
<td>UpdateLayoutTree</td>
<td>elementCount, frame</td>
</tr>
<tr>
<td>Rendering</td>
<td>Layout</td>
<td>dirtyObjects, totalObjects, elementCount, frame</td>
</tr>
<tr>
<td>Rendering</td>
<td>UpdateLayer</td>
<td>layerId, layerTreeId</td>
</tr>
<tr>
<td>Rendering</td>
<td>InvalidateLayout</td>
<td>nodeId, stackTrace, url, scriptId, columnNumber, lineNumber, frame</td>
</tr>
<tr>
<td>Painting</td>
<td>Paint</td>
<td>clip, frame, layerId, nodeId</td>
</tr>
<tr>
<td>Image</td>
<td>Decode Image</td>
<td>imageType, Image</td>
</tr>
</tbody>
</table>

Table 6.1: Top-10 events and their metadata.
counter, etc.), are logged for all events, some may also include additional event-specific metadata. The format of the latter metadata differs across event types and, at a high-level, it captures information about the specific nature of an event call\(^2\). In Table 6.1, we list the metadata types for the top-10 events called during the page loads for Alexa top-1K websites\(^3\), and they encode information such as URL of a network fetch (url), JS function identifiers (functionName, scriptId) and code coordinates in JS file (fileName, columnNumber, lineNumber), number of nodes touched in layout (dirtyObjects, totalObjects, elementCount), area painted (clip), etc.

Additionally, the logging infrastructure can be tuned to generate further low-level events such as code coverage, i.e., the bytes of unused JS code [113], heap allocations [11], process memory dumps [137], etc. Note that, the default logging tools, such as Timeline Profile [491], do not natively log this additional information and special flags need to be configured to generate these domain-specific event logs. In summary, the logging infrastructure provides a wealth of raw low-level logs, accounting for both the type and specific nature of events, and these logs can be leveraged to gain low-level visibility into the browser mechanics during the page load.

### 6.2.5 Chromium Memory Allocators

While loading the page, Chromium uses several different allocators for allocating memory to any of its processes or components. Table 6.2 lists all such allocators and their involvement during the page load process. While some of the allocators have a distinct one-to-one mapping with page load actions, e.g., net and V8 are always used for network requests and JS evaluation respectively, others such as malloc, partition_alloc and shared_memory are used across the different page load actions. Furthermore, the memory allocated as a result of executing a JS function may span multiple allocators, since JS APIs often interact across browser components, such as compositor or GPU, for a variety of actions, such as networking, DOM manipulations, animations, etc., that may lead to memory allocations to store DOM, media objects and render/layout/style information.

To illustrate, let us consider a code example (Listing 6.1) where a function `bar()` uses jQuery to iterate over boxes in HTML and modify their text and style. Several browser components and allocators are involved during `bar()`'s execution and the listing annotates the scope of JS interactions and list the related browser events triggered by the JS code line. While parsing and compiling the code, V8 emits events, such as compile and FunctionCall, and the memory overhead of such tasks is dynamic, depending on the bytes of code, size

---

\(^2\)The documentation for the different event types and their metadata is spread across multiple sources and can be found here [496, 141, 83].

\(^3\)Events are not natively classified into “classes” and we use Chrome performance reference [141] to assign the classes.
of included library, types of APIs, types and size of runtime objects defined (e.g., boxes array), etc. When the jQuery APIs (text and width) interact with DOM to modify text node and style, rendering engine emits events, such as InvalidateLayout, ScheduleStyleRecalculation and UpdateLayoutTree [492], the compositor allocates memory to update layout and generate the tiles on screen, and the memory overhead is variable with respect to the type of content added and the complexity of DOM changes (as we later show in § 6.7). Similarly, if any network fetches were involved, the overhead of storing the objects would highly depend on the type and size of the respective objects.

As JS DOM API calls [337] require access to the DOM, Chromium stores the references to DOM nodes in the JS heap (V8 heap [273]), while the actual objects and metadata of DOM nodes are stored within the Renderer process [272, 11, 273], e.g., reference to an image is stored in DOM while the actual image and its bitmap representation are stored in the Renderer. A key consequence of this design and the cascading effects of JS is that allocations and browser work across different browser components need to be observed, in order to truly account for the overhead of JS.

6.3 Motivation

In this section, we present a longitudinal measurement to show that the web memory overhead has consistently increases over the years and JS, a cornerstone of the web ecosystem [43, 236], is a dominant and rising factor behind the overhead. We further present challenges with JS memory inference and highlight limitations with traditional memory measurement techniques.

6.3.1 Longitudinal Study of Memory Usage

To investigate how the memory state of the web has evolved, we analyzed the memory utilization for top 250 popular Alexa websites\(^4\) over a period of 6 years, from 2015 to late 2021. We used the WayBack Machine from Internet Archive [44] to measure the page load memory\(^5\) for the historical snapshots of the respective websites. For each website, we captured the recorded snapshots from each quarter of the year\(^6\), randomly

\(^4\)Alexa list collected in August 2021.

\(^5\)We define “page load memory” as the max memory consumed by a browser tab, measured using meminfo [217] tool.

\(^6\)83% of the websites had at least one snapshot per year, while the rest were absent in one or more years.

---

Listing 6.1: JS cascading impact illustration.
select one snapshot per-quarter, and repeat the measurement ten times. This process was repeated for every year, from 2015 to 2021, and the webpages were loaded on an android mobile phone (test-bed discussed in § 6.5.3). We also measured the ground-truth JS memory, defined as the memory single-handedly contributed by the JS code (methodology discussed in § 6.6). To validate that the webpage snapshots from WayBack Machine are representative, we measured the memory of the actual websites (i.e, live websites in 2021) and compare live memory with the 2021 snapshots. Comparing a live webpage with its historical snapshot, we observe that the memory for snapshots is within 4% margin of error for the median website (9% at the tail)7.

Figure 6.1 plots the average memory per-year and shows a consistent increase in memory over the years. Comparing the trends for the two memory metrics, we observe that the dominance of JS memory has increased with time: while JS contributed 28.6% (16.3MBs out of 57.1MBs) of total memory in 2015, its contribution stands at a staggering 64.8% (49.4MBs out of 76.3MBs) in 2021. Interestingly, we observe that websites with the highest total memory overhead (in 150-220 MBs range) tend to have the highest JS overhead (58.5%-77% of total memory), highlighting that optimizing JS memory overhead is the key factor towards reducing the overhead of memory-bloated websites. In Figure 6.2, we present the percentage increase in JS memory (using 2015 as the baseline) for webpages that are consistently present in the snapshots every year and show that the trend for JS memory holds, with ~67% increase at tail in 2021. Our measurements show that JS memory overhead has significantly increased over the years and, today, is a key dominant factor behind web’s memory overhead. Consequently, it is imperative to understand JS memory usage, in order to fix web’s memory plights.

6.3.2 Root-cause of JavaScript Memory and Attribution Challenges

As discussed in § 6.2, JS often leads to cross-component interactions, leading to direct (i.e., V8) and indirect (i.e., components outside V8 boundary) memory allocations, thus requiring nuanced visibility into the JS and browser mechanics to identify the source of overhead. Traditionally, browser work, i.e., browser events called while loading the webpage, is used as an indicator to understand the performance overhead, e.g., Chrome

7Since JS may dynamically construct URLs for web resources, a historical snapshots may request for URLs (e.g., images) that may not be valid anymore and lead to the small margin of error in the memory measurements. A recent work [206] reports that the impact of such invalid fetches is modest at median (5% of network bytes at median) and more significant at tail (45% of bytes).
guides on heap profiling [62, 272] define low-level V8 events, such as parse/compile and object allocations (e.g., string, arrays, native objects), as the source or root-cause of heap memory. Since our scope is not limited to just V8, we extend this definition and define the root-cause of JS memory allocations as “the set of browser events, together with their corresponding parameters, triggered across components as a result of executing a JS function”. Intuitively, this definition captures the cross-component browser work, done on JS’s behalf, that incurs the memory allocation overhead. While the “events” identify the type of components/work involved, the “parameters”, encoded as event metadata (§ 6.2.4), provide visibility into the nature of browser work.

Below, we discuss the key challenges in attributing the root-cause:

- While loading a webpage, the browser executes thousands of JS functions (∼2800 at median for Alexa top 1K websites) and emits tens of thousands of fine-granular browser events. In Figure 6.3, we plot the duration between two consecutive events for top 1K websites and the 10s of millisecond duration at median shows the fine-granularity of browser events. In fact, a majority of such events (>70% for median top 1K websites) are precipitated by JS and are outside scripting (i.e., V8) scope: Figure 6.4 plots the percentage of categorized events triggered by JS and we observe a significant percentage of layout, painting, loading and layer events. While the V8 event metadata encode information to tie them back to JS functions, such mapping are not available for most non-V8 events (e.g., Layout, Paint, Decode Image, etc.). These trends present two challenges: first, fine-granular memory measurements are required to account for the cross-component allocations made in response to executing a JS functions, and, second, specialized techniques to attribute events to JS functions are required to attribute the direct and indirect root-causes.

- Assuming that function-level memory and event mappings are available, the hundreds of events in a function may not be equally be responsible for the memory overhead. Chrome guide on profiling page load memory [137] recommends a check-pointing strategy where memory is measured at timestamps \( t \) and \( t_i \) and events called within the duration are considered as potential culprits. However, since memory overhead may be sensitive to different event types/parameters (e.g., code size for parsing), statistical techniques, such as correlation or causality, are required to identify root-cause events that contribute to memory in a statistically significant way. Further, the scope of analysis techniques may vary based on the nature of a function's root-cause events, e.g., for functions whose events do not span outside V8, the overhead can be understood by primarily focusing on the V8-centered events, whereas others may also require broadening the scope to network transactions, compositor or GPU.

In light of the challenges, the ideal measurement technique is required to capture fine-grained, cross-component memory allocations, as well as, attribute the key root-cause of the memory overhead. Such a technique equips a developer with a function-level dissection of memory, thus enabling them to, first, identify the problematic functions and, second, connect the function JS code to low-level browser events that are invaluable for understanding the cause of overhead and defining the scope of required optimizations.

Next, we discuss the existing tools for measuring JS memory and their limited prowess in capturing the indirect memory or attributing the root-cause.
6.3.3 Limitations with Existing Tools

There are two broad classes of tools for measuring memory: OS-level [217, 154, 405] and Browser-provided [491, 137, 11] tools. In Figure 6.5, we plot the JS memory coverage, i.e., percentage of ground-truth JS memory a tool is able to account for.

Timeline Profile and HeapProfiler are based on Chromium: the former captures a time-series of JS heap and browser activity (e.g., network, rendering, scripting events), while the latter captures V8 heap memory broken down by JS functions and objects (e.g., strings, arrays). We observe low coverage (below 30% at median) for both, the key reason being that these tools are only limited to V8 heap and do not capture the indirect JS memory. In Figure 6.6, we plot the percentage of JS memory contributed by the various allocators and observe that a variety of non-V8 allocators, such as cc or compositor, malloc, etc., make significant contributions, with V8 contribution at only 27% at median. Our conversation with the Chrome developer indicated that although the report generated by HeapProfiler includes DOM node references, the tool itself does not measure the memory allocated to objects being referenced and, thus, only captures a subset of the overall memory footprint.

On the other hand, MemoryInfra and OS-based tools provide full coverage: the former resides inside the browser and captures all allocators, while the latter (e.g., meminfo [217], dumpsys [154] and perfetto [405]) capture information at the process level and provides an aggregated memory view. Although these tools do not natively measure JS memory, recent work [12] used such tools to capture JS memory by removing all JS code from a website and taking a difference between vanilla website’s memory (i.e., with all the JS code intact) and the stripped-off version. While these tools provide good coverage, they are limited in their ability to localize the source to specific functions and identify the root-cause of the overhead. Since OS-based tools capture information at the process level, they do not have any visibility into the functions being processed and the corresponding browser mechanics, and thus provide coarse-grained information. Similarly, MemoryInfra measures memory at a 10s granularity by default and only provides an after-the-fact memory snapshot, i.e., total allocator-wise breakdown of memory consumed due to the browser work in the preceding 10 seconds. Though one can modify the MemoryInfra code to increase the frequency, its major deficiency is that it only localizes the overhead to coarse-grained allocators and does not capture fine-grained memory for individual
functions. Further, it lacks the ability to connect the cascading events to the source function (i.e., root-cause events).

Thus, while the existing tools are able to identify the symptoms, i.e., how much memory is allocated?, they fail in attributing the source of this symptom due to their coarse-granularity and lack of event attribution to functions.

6.4 JS Capsules

The key insight behind our methodology is that JS engine in browsers\(^8\) execute JS code in a single-threaded fashion [518, 330, 376] per frame and use a single, main, thread for scheduling JS and most rendering events [330]. This serial-access model simplifies page development and provides safe concurrency guarantee for the shared DOM data structure. More explicitly, the model guarantees that a renderer process never executes more than one JS function in parallel and guarantees a sequential order for the rendering events, stimulated by a JS function. We leverage this guarantee to generate isolated blocks of code, compute their isolated memory footprint and attribute browser events to the JS functions.

Let \( F = \{ f_1, f_2 \ldots f_n \} \) be a set of JS functions in a website and \( E = \{ e_1, e_2 \ldots e_m \} \) be the set of cascading

\(^8\)Also applicable to non-chromium browsers such as Nitro JS engine in Safari.
events, such that $f_i$ triggers set of events $E_i$, where $E_i \in E$. Assuming that a continuous timeline of memory measurements exists, captured at a fine-granularity (tens of ms), we can calculate a function’s raw memory $(raw\_mem(f_i))$ by taking a difference of two memory snapshots, due to the safe concurrency guarantee.

\[
raw\_mem(f_i) = mem\_timeline(ts\_end_i) - mem\_timeline(ts\_start_i)
\]  

(6.1)

Where $ts\_start$ and $ts\_end$ represent the timestamps when a $f()$ execution started and ended, respectively, and $mem\_timeline()$ outputs the memory state at the given timestamp. However, $raw\_mem(f_i)$ only captures the true memory of $f_i()$ if $f_i$’s cascading events ($e_i \in E_i$) are contained within $ts\_start$ and $ts\_end$. Otherwise, in order to capture the memory overhead of cascading events ($E_i^*$) the occur after $f_i$’s execution, $f_i$’s actual memory cost is defined as the sum of memory allocated during executing the function, as well as any cascading events out of $f_i$’s interval ($E_{i*}$).

\[
mem(f_i) = raw\_mem(f_i) + \sum_{e_i \in E_i^*} raw\_mem(e_i)
\]  

(6.2)

Where the event’s $raw\_mem()$ is calculated similarly to the function’s. Given this terminology, we define JS-Capsule as a set of one or more JS functions, represented by a unique, non-overlapping time interval (i.e., start and end timestamps). The time interval encapsulates one or more JS functions and provides the basis for decomposing a website’s JS into isolated blocks and we discuss the rationale behind the design in § 6.4.1. Given a capsule $C_i$ encapsulating functions $F_i$, where $F_i \in F$ and $E_i \in E$ is the set of $F_i$’s outlying cascading events, $C_i$’s memory is calculated as:

\[
raw\_mem(C_i) = mem\_timeline(max(ts\_end_i \forall f_i \in F_i)) - mem\_timeline(min(ts\_start_i \forall f_i \in F_i))
\]  

(6.3)

\[
capsule\_mem(C_i) = raw\_mem(C_i) + \sum_{e_i \in E_{i*}} raw\_mem(e_i)
\]  

(6.4)

Figure 6.8: Browser mechanics for executing $bar()$ JS

Figure 6.9: JS-Capsules illustration

Figure 6.10: JS-Capsules are created by identifying JavaScript events and appropriate cascading events, then using their timestamps to capture the associated memory allocations.

### 6.4.1 JS-Capsule Design and Illustration

Next, we discuss the details of how we capture the four key pieces of information for generating JS-Capsules, i.e., (i) generating fine-grained memory timeline, (ii) capturing JS execution traces and browser events, (iii)
attributing cascading events, and (iv) generating JS-Capsule boundaries.

**Capturing memory:** We use Chromium’s requestMemoryDump API [135] for capturing a continuous memory snapshot timeline\(^9\). Each snapshot comprises process’s current memory (private footprint bytes field) and allocator-wise distribution (allocators field) on per-process basis, i.e., isolated timelines for Browser, GPU and each Renderer process\(^10\). For each snapshot, we set deterministic to true in the API that enables deterministic results by forcing garbage collection [136]. Note that garbage collection is non-deterministic and may have reclaimed memory two snapshots, that can lead to discrepancies in memory measurements. To eliminate garbage collection-related non-determinism and keep measurements consistent, we force garbage collection before every snapshot. This calibration step minimizes the non-determinism of garbage collection call during two snapshots and is validated in Section 6.6.

**Capturing JS execution traces and browser events:** We use Chromium’s Tracing API [134] to capture low-level events during a page load. The Tracing API defines a number of different categories, among which v8.cpu.profile captures JS execution traces. For functions defined by the developer in HTML/JS files or referenced from included libraries, v8.cpu.profile trace include JS timing information (start and end timestamps) and metadata (function name, file URL, unique IDs assigned by V8, location of function code in file, etc.). We include all relevant event categories while taking the trace to capture cascading events such as loading (ParseHTML, FinishLoading etc), scripting (EvaluateScript, FunctionCall, v8.compile, etc), rendering (InvalidateLayout, Layout, RecalculateStyle, etc), paint (Paint, CompositeLayers, ImageDecode, etc), among other categories. We collect all the categories and events mentioned in Chrome’s performance reference documentation [141].

**Attribute cascading events:** We leverage a function’s timing information (i.e., start and end execution timestamps) to attribute events. To illustrate, Figure 6.8 presents the browser mechanics for executing the function bar() from Listing 6.1. The figure shows the call-stack for functions called within bar(), their timing information and browser events (e.g., recalculate style). Two time statistics are reported for each function, f(). The self time represents the time spent in f() and the total time represents the time spent in f() plus its dependents functions. In Figure 6.8, we annotate the self and total time for bar(). The total time for bar() includes the time used by the dependent functions (e.g., jQuery functions text() and width()) used by bar().

When a browser event is triggered as a response to a JS call (e.g., recalculate style in Figure 6.8), one of three scenarios can happen: (i) the self-time for the event may overlap with f()’s total-time, (ii) the event may occur after f(), i.e., does not overlap with f()’s total-time, or (iii) the event may occur outside of main thread context (e.g., GPU events, events for creating bitmaps of tiles to paint that chromium sometimes offloads to raster/composite threads [330]). While the solution for the first scenario is straight-forward, i.e., events contained with the start and end timestamps are attributed to the respective JS-Capsule, the other two present a challenge. Specifically, while the single-threaded browser mechanics for the main-thread ensures an ordering, some cascading events may occur outside of JS-Capsule duration or in another browser thread/process (we call such events out-of-duration events and are mostly either layout or paint). For out-of-duration events, we adopt a heuristic that is used in task attribution for leading web performance diagnosis tools such a

---

\(^9\)Memory snapshot collected every 50ms.

\(^10\)The browser spawns an isolated renderer process for each of the cross-origin iframes present in the website [335].
The heuristic is based on the observation that renderer process emits scheduling tasks, e.g., ScheduleStyleRecalculation or InvalidateLayout etc, indicating which function f() is responsible for the subsequent out-of-duration events. We use this heuristic to attribute such events to the subset of preceding JS-Capsules that occur after the last occurrence of the respective out-of-duration event and contain a respective scheduling task. This inference is applicable for both renderer’s out-of duration events and events on other process (e.g., GPU).

Generating JS-Capsule boundaries: From a design perspective, there are two extremes for defining a JS-Capsule’s boundaries: either use bar()'s total-time for marking the JS-Capsule boundaries and aggregate the dependent functions and events, or use each f()’s self-time to generate one JS-Capsule per f(). While the second approach is more fine-grained, it requires capturing memory at a very high frequency. Yet, JS functions are short-lived, e.g., self-time for style() and set() is 3.52ms and 0.37ms, and Chromium APIs for capturing memory snapshots operate at a coarser granularity. Due to this practical limitation, we adopt the first approach and our algorithm to preserve find-grained information is as follows:

We climb the call-stack bottom-up and, for each function f(), inspect if its memory can be measured in isolation, i.e., the memory difference between the end and start of f() does not overlap with any subsequent function g() or events that are not f()'s dependent. If possible, we use f()'s total-time (i.e., start and end timestamps) for representing a distinct JS-Capsule. Otherwise, we continue the search until we reach the top call (bar() in Figure 6.8). In the case when such a measurement is not possible for the top call, we merge the timing information for consecutive function calls and encapsulate them in a single JS-Capsule. However, in practice, we observed that this merging was only required for a small proportion of functions and in Section 6.6, we further discuss some calibration steps and their impact on our design.

6.4.2 JS-Capsules Generation and Structure

To generate JS-Capsules, we load the given website on a mobile browser and generate the logs through APIs discussed in § 6.4.1. Once the logs are collected, our Python-based framework implements the JS-Capsules algorithm to generate the JS-Capsules. A key challenge in log processing was the lack of documentation available for log structure and meaning. We perused Chromium’s tracing code to understand the structure and the log-processing framework is implemented in 3.2K LoC. We plan to open-source the code to help the community in performance analysis.

Each JS-Capsule is represented as a key-value pair, mapping a custom unique ID for the JS-Capsule to a set consisting of memory size, dependent functions, browser events and metadata. We also log the dependency information between the functions and browser events on a per-function basis, i.e., a nested dictionary that maps JS-Capsule-ID to f()-ID and f()-ID to browser events and metadata. Since functions may have repetitive or anonymous names, we use the V8-generated unique IDs as f()-ID. We also store a wide range of metadata for each function, such as the URL of the JS file that contained the f(), HTTP headers for the respective URL, f() code content and DOM APIs called within f(). To track JS API calls, we wrote a generic JS framework that leverages JS prototyping [339] to track API calls and their acted DOM node.

11While total-time boundaries for the different f() may overlap with each other in Figure 6.8, self-time boundaries are always non-overlapping due to single-threaded nature of JS execution.
At a high-level, this framework adds shims around all JS DOM API calls listed in [337] and tracks function arguments and the nodes manipulated by a call. This framework is injected to the website code through a proxy in our testbed and communicates the results back to a data-collection server. For DOM-centered analysis, we further use Chrome’s DOMSnapshot API [133] that captures the entire DOM tree data-structure for a webpage.

The JS-Capsule-generation procedure is performed for each renderer process\(^\text{12}\), since the single-thread browser mechanics for the main-thread are only limited to a single renderer process scope. The JS-Capsules for a website are represented as a dictionary, mapping process-ID to the list of JS-Capsule-IDs.

### 6.5 Measurement and Analysis Methodology

In this section, we discuss the analysis techniques that we use for explaining measurement results in § 6.7 and present the implementation details of our mobile phone test-bed.

#### 6.5.1 Clustering and Explaining Diverse Capsules

A key goal of our paper is to leverage the JS-Capsules to characterize JS memory for a diverse range of websites, e.g., Alexa top 1K. While JS-Capsules decompose the memory in isolated JS blocks, studying the hundreds of thousands of capsules can be an exhaustively tenuous task and requires techniques through which we can compare and contrast the memory dynamics of different capsules and draw generalizable trends.

Our key intuition behind drawing generalizable patterns across capsules is that “while the nature of JS functions may vary across websites, there is a limited set of browser allocators involved in processing an arbitrary function.” Consequently, in order to draw a universal set of memory allocation patterns, the different capsules can be clustered on the basis of their allocator fingerprints, i.e., memory contributed by the different types of allocators for a capsule. The key advantage of using allocator memory fingerprints is that it directly identifies the culprit browser components, e.g., dominance of \(v8\) allocator for a capsule indicates that \(V8\) component is the key memory contributor, therefore enabling us to narrow our focus to specific browser

\(^{12}\)Multiple renderer processes may be spawned if the website has multiple iframes.
component(s) for a cluster. Thus, based on this intuition, we first cluster the allocation memory makeup of capsules (Allocator-based Clustering) and then, for each memory allocation pattern (i.e., cluster), further employ two techniques to (i) identify the source/nature of capsule JS (JS Classification), and (ii) per our root-cause definition from § 6.2, identify the key browser events that contribute the memory overhead (Key-events Inference).

Next, we describe the details of these techniques.

**Allocator-based Clustering:** We represent each capsule by their allocator-wise memory breakdown, i.e., the MBs of memory allocated by the different allocators shown in Table 6.2. We experimented with a number of clustering techniques including Kmeans, SVM and hierarchical clustering, and empirically evaluated Kmeans to be the best option. For the choice of “K” in Kmeans, we tested a range of values between 2 to 50 and used the *elbow method* [465] for analyzing K’s impact on sum of squared errors (SSE). We found the elbow to be K=26 (Figure 6.11), representing the point after which the change in SSE smoothens and, consequently, we use 26 as the total number of clusters. This classification enabled us to identify the key memory allocation patterns observed for the capsules across websites and summarized the hundreds of thousands of capsules into a handful of memory patterns.

**JS Classification:** We classify the nature of JS-Capsule functions based on the following properties: origin, cross-origin, advertisements, trackers and libraries (e.g., jQuery, Ajax). To identify the origin source, we inspect the Sec-Fetch-Site field in the HTTP headers that indicates the relationship between a request initiator’s origin and the origin of the requested object [526]. For ads/trackers, we use a publicly available ad/tracker-blocking lists [434, 547] (collected in July, 2022) and classify the source’s URLs by feeding the list to the **adblockparser** library [348] which uses regex matching for inference.

Contrary to origin, ad or tracker, classifying a function into libraries is more challenging. While the source URLs provide hints, e.g., library name in URLs such as `code.jquery.com/jquery-3.6.0.min.js`, there is no strict guarantee and developers often host library’s code in arbitrary files, e.g., drupal.org includes jQuery and CreateJS code in a random file under `drupal.org/files/advagg.js/`. Thus, simple URL-based matching is bound to miss such appearances. To tackle this challenge, we developed a database of all JS functions included our corpus of the top 1K websites. Taking inspiration from prior work [99], we generated unique fingerprints for each function and inspected whether a function appears across multiple websites. We make the assumption that the appearance of the same function across multiple websites (we use a threshold of 5 websites) indicates that it is part of a shared library. While computing the frequency of a function across multiple websites, we ensure that only the unrelated websites contribute to the frequency, e.g., functions repeated across `google.com` and `google.co.uk` (or `apache.org` and `apachefriends.org`) only contribute once. We manually inspected the hostnames to mark such duplicate websites.

**Key-events Inference:** Per the design details discussed from § 6.4, the memory of a capsule represents the cumulative overhead of its constituent browser work. Since all such events in a capsule may not be equally

---

13While the event composition of a capsule can be directly used for this purpose, such clustering does not account for the memory differences for the different types of events. We further faced a bias-ness issue with event-based clustering, further discussed in § 6.10.2

14While different text structural features or API features can be leveraged for computing fingerprint, we use *md5 hash* for the function code content due to its simplicity and fast computation.
responsible for the memory overhead, we used statistical and machine learning techniques to identify the key events that are the overhead root-cause.

While Chrome guides generally suggest correlation to identify the key events [137], we tested a wider range of techniques including feature correlation [26, 362], causality [455], and feature importance through models such as Random Forests (RF) [449, 424, 317, 42] or k-means [436]. We found RF feature importance as the most appropriate fit and further conducted error estimation, correlation and causality tests to validate this selection. At a high-level, this technique uses “the relative rank (i.e., depth) of a feature used as a decision node in a tree can be used to assess the relative importance of that feature with respect to the predictability of the target variable” [424, 317]. For each dimension, i.e., feature, a metric called mean decrease in impurity (MDI) is calculated that quantifies the importance of the respective dimension in predicting the final outcome (in our case, allocator memory of the capsule). Features whose importance is measured to be above a manually-defined threshold are selected as the key features. The complete list key events for each cluster is presented in Table 6.3 and the results of error estimation, correlation and causality tests to validate key-event inference is discussed in § 6.6.

6.5.2 Analysis Workflow

Next, we describe the end-to-end workflow used for analyzing the measurement results.

The workflow begins with the generation of capsules for the set of targeted websites (Alexa top 1K for our study). Next, we analyze the allocator-wise memory breakdown of each capsule in the data-set to identify the allocators that dominate the memory overhead. The key significance of this step is that if the overhead is heavily swayed towards a single (or set of allocators), we can narrow our focus on the respective allocators (and the browser components that use them) for explaining the capsule’s memory. As minor differences may exist within capsules that share a similar dominant allocator, next, we leverage the allocator-based clustering to highlight such differences. This clustering is especially helpful for the cases where a capsule may involve minor allocations from a variety of allocators. The next step in our analysis pipeline is the extraction of key events for each cluster that directly identify the key browser mechanics behind the memory overhead. The final step in the pipeline is to leverage the events and their respective metadata and use statistical techniques to analyze the relationship between event/metadata and memory. However, the scope of such analysis highly depends on the nature of events and browser components involved. For instance, for capsules where the key events are contained within a single component (such as V8), the analysis of the events and metadata (such as compiled code size and object allocations) is enough to directly attribute and explain the memory. Similarly for network events, the overhead for storage can be directly explained by the respective event metadata (such as type and size of objects). However, for more complex capsules, such as ones that involve rendering due to JS-DOM interactions, understanding the overhead is not always straightforward due to the cross-component interactions. In such cases, we use statistical techniques (correlation) and further employ targeted experiments to test and validate the observed trends.

To further complement the root-cause events and metadata, our analysis pipeline further leverages three pieces of information: (i) our JS classification framework to classify the source of JS functions in a capsule, (ii) our JS API logging framework to identify the APIs called in a function that precipitate the root-cause
events, and (iii) Chrome DOM snapshots [133] to gain visibility into the DOM structure of websites. Note that, the DOM snapshots are only used for understanding the nature of DOM trees for different websites, whereas the DOM interactions triggered by a capsule are directly measured from the relevant event metadata, such as dirtyObjects, totalObjects and elementCount fields from Layout events (Table 6.1).

6.5.3 Measurement Testbed

Our testbed comprises three Android phones and a dedicated desktop to automate any activity on phone (e.g., open or close browser, clearing app cache, etc) through Android Debug Bridge (ADB) [129]. To interact with Chromium APIs for log collection or load webpages, the testbed uses chrome-remote-interface [114] (implemented in node.js) and does not require device root access.

We use three phones: QMobile Q Infinity B (low-end, 1GB RAM, Quad-core 4x1.2 GHz CPU, 4.95in screen, 217ppi), LG G5 (mid-end, 4GB RAM, 2x2.15, 2x1.6 GHz CPU, 5.3in screen, 554ppi resolution), and Samsung Galaxy S22 (high-end, 8GB RAM, 1x2.8, 3x2.5, 4x1.8 GHz CPU, 6.1in screen, 425ppi resolution). The devices run Android 7.0 OS and Chrome v91 is used for our measurements, with no background applications or tabs to ensure that there is no background memory contention and all memory is available to the page load process. To prevent any cross-measurement contamination, a new browser instance is spawned and its application cache is cleared before each trial. The browser is terminated after each trial and each measurement is repeated five times. The phones are connected with an energy source to keep them always fully charged. We also ensure that the page fully loads by tracking the onLoad event.

To ensure reproducibility, the desktop hosts mitmproxy [166], a popular record-and-replay tool, to record webpages and later replay them to the mobile devices. The last-mile link (i.e., the device access link) is set to home networking conditions (100Mbps, 20ms RTT) during replay. A key advantage of using mitmproxy is its event-driven scripting ability to modify web objects on-the-go, i.e., a Python script can inspect the request/response objects at runtime and modify the response as required. We use this scripting ability to inject our JS API tracker framework. We conducted measurements for the Alexa top 1K websites (list collected in April, 2022). Our list provides good coverage over different website categories (inferred through Cyren tool [120]) such as news (131 websites), shopping (47), social network (25), sports (13), search (41), business (165) and entertainment (87). We only focus on the landing pages and do not interact with the webpage (e.g., scroll) to focus our measurements on only the base JS memory. Further, our devices load the webpages in their default language, i.e., there is no JS overhead of translating the webpages to English.

6.6 Validation

In this section, we validate a number of our measurement and design choices.

**Memory coverage and overhead:** In order to validate memory coverage, we compared the sum of memory across all the JS-Capsules in a website with its ground-truth JS memory. Since the existing tools are unable to isolate JS memory across various allocators, we calculate the ground truth by removing JS code from a webpage and taking the difference between the memory observed for the default webpage against
the modified page with JS-removed\textsuperscript{15}. Our key insight behind this methodology is that with JS removed, the allocators would not incur the memory overhead of executing JS (and cascading effects). Additionally, to verify that our logging techniques do not introduce overheads or noise into our memory measurements, we disable logging and snapshotting while computing ground-truth memory and use the dumpsys\textsuperscript{154} for measuring the memory for the browser processes. We remove all the JS code from the website by emptying the JS files and script tags in HTML (both inline code and src field). Note that, we still keep the script tags in

\textsuperscript{15}We validate JS removal by ensuring that no scripting related events are observed and the total JS scripting time is 0.
the DOM to keep the DOM structure intact. To calculate the total memory, we first aggregate the timelines of individual process memories (i.e., bin the pss values by time and take sum) and take the max of the aggregated timeline.

Figure 6.12 plots the memory coverage for 100 randomly selected websites for the three phones. We are able to capture >85% of ground-truth JS memory for more than 90% of the websites – illustrating good coverage. For the cases where coverage is below 95%, we observe two key reasons behind the coverage gap. First, since we force garbage collection before every measurement to eliminate non-determinism with memory measurements, a consequence of this design choice is that the browser aggressively reclaims unused memory (in contrast to the ground-truth measurement) and this design choice results in our framework having slightly lower coverage values than the ground-truth. Second, since we do not know when a function execution would end apriori due to browser non-determinism, there is no formal guarantee that the capsule boundary ends with the end of function/event. Assuming a capsule ends at $t$ and we have memory measurements available at $t - \delta$ and $t + \delta$, using the former reading as the final snapshot can underestimate the memory, while the latter can over-estimate (due to potentially capturing memory unrelated to the capsule). We take two calibration steps that result in coverage gap: (i) we only select the $t + \delta$ reading as the final memory snapshot for the capsule if no unrelated function/events are called between $t$ and $t + \delta$ and, otherwise, select the $t - \delta$ reading to prevent over-estimation, and (ii) for brief out-of-duration events (order of ms) where memory cannot be measured in isolation (e.g., due to presence of unrelated events within $t$ and $t + i$), we exclude the memory to prevent contamination due to unrelated events. Both the forced garbage collection and the calibration steps result in under-estimation of capsule memory, however as the results show, the impact is minor (within 3.5-7.5% at median) and up to 23% at the worst-case. Figure 6.12 also shows slightly different memory coverage for the different devices and we discuss them next.

Impact of device type on capsules: Figure 6.13 plots the number of capsules observed across websites for the three devices. A varying number of capsules for the different websites is expected, as websites differ in the nature and extent of client-side JS usage. However, we also observe that the number of capsules reduce as we move from a low-end (Q Infinity) to the high-end (S22) device. Higher CPU clock frequencies for the mid/high end phones results in faster processing, leading to JS functions (or events) taking fewer milliseconds to execute as compared to the low-end phone. Since our bottom-up search algorithm for generating capsule boundaries relies on a function’s timing duration for isolating memory overhead, faster function executions (i.e., lower total-time) leads to grouping a higher number of adjacent functions together in a capsule, leading to overall a lower number of capsules. We further conducted an experiment where we slowed the clock frequency to half its original value and, as shown in Figure 6.13, the slower frequencies lead to a higher number of capsules, especially for the high/mid end devices. Consequently, we observe the clock speed of a device to directly impact the fine-granularity of our technique, as we take memory snapshots at a constant frequency and inflated function execution times (as well as load times) lead to a higher number of snapshots that improves the efficacy of bottom-up search algorithm for decomposing the isolated JS/memory blocks.

Function distributions across capsules: Next we discuss the number of functions grouped in a capsule, focusing only on the low-end device where our framework results in highest number of capsules, i.e., finest granularity among the devices. To measure the number of JS functions encapsulated by a capsule, we sort
JS-Capsules across a website based on the number of functions and Figure 6.14 plots the median, p75 and p95 number of capsules in a website. Multiple functions per capsule is expected as our design merges dependent functions into a single capsule to tackle practical limitations, as discussed in Section 6.4.1. For the capsules encapsulating a high number of functions, we observe that most functions (up to 80% at median) to be short-lived, taking less than 1ms to execute.

Taken together with Figure 6.13, our framework results in decomposing a website’s JS at a reasonably fine-granularity of capsules in order of hundreds for more than 80% of websites. As we later show in § 6.7, this granularity is sufficient for providing semantic meaning to JS memory, dissecting the different root-causes, and uncovering the potential avenues for optimizations.

**Measurement noise:** In Figure 6.15, we compare the total JS-Capsule memory for 100 randomly selected websites with and without forced determinism (i.e., set deterministic field to control garbage collection before measurements). The figure plots the standard deviation for 10 repeated measurements and shows that our design choice highly reduces the noise in measurements and improves the accuracy of our framework.

**Key feature inference:** Next, we validate the effectiveness of our analysis framework for identifying the key events. Figure 6.16 plots the Random Forest prediction error (model discussed in § 6.5.1). The low error (0.32 at median at worst) shows that the RF model accurately models the mappings. Even for the tail at the worst case, we observed 1.18MB error and that accounted for less than 4% of the total capsule JS memory. Figure 6.17 plots CDF of Spearman’s coefficient for correlation between a capsule’s events and memory. We first cluster a capsule into its cluster, correlate the count of key events with memory, and the CDF plots values across all the capsules in our data-set. We observe moderate-to-strong correlation for most capsules (ρ=0.68 with p-value < 0.000 at median). The key reason why some events show lower correlation is that event metadata (e.g., number of bytes of JS code parsed, number of nodes layout, etc.) directly impact the browser work for an event and as such combining the key event with their metadata leads to further strengthening the relationship (as we further show in measurement results).

**Library inference:** To validate our library inference technique, we generated a list of libraries which contribute atleast 5MB of memory to the 100 websites. We randomly selected 50 of these libraries and manually inspected the function code and validated that our inferences are accurate.

### 6.7 Measurement results

In this section, we present the measurement results for the Alexa top 1K websites. Leveraging our analysis methodology framework, we analyze the memory allocation patterns for capsules, explain the root-cause of memory overhead, and discuss the key aspects that influence JS memory overhead. Per the discussion in § 6.6, the different device types can have a impact on the granularity of the capsules and our memory characterization in this section focuses on the device with the finest granularity, i.e., the low-end device.

---

16 We use Spearman correlation due to its ability to assess monotonic relationships, while other coefficients like Pearson’s are more suited for linear relationships [362].

17 Alexa list collected in June 2022.
6.7.1 Clustering Capsules’ Memory Allocation Patterns

Analyzing the allocator-wise memory breakdown for the capsules across the 1K websites, we observed that, for a significant proportion of capsules (numbers presented in Table 6.3), the memory overhead is heavily tilted towards a single allocator. The key significance of this trend is that the bulk of memory overhead for such capsules can be pinned down to a single allocator, and by extension, a component of the browser\(^\text{18}\). Consequently, based on the key allocator that contributes the memory, we grouped the capsules into 5 sets\(^\text{19}\): V (key allocator v8), R (cc or compositor which is responsible for rendering), G (gpu), M (malloc) and X (variety of allocators). Figure 6.18 presents the clusters observed for the capsules and the bars plot the allocator-wise JS memory for the cluster centroid. For the 26 clusters, where K=26 is the number of K-means clusters for the data-set calculated using the elbow method, we observe that 15 out of the 26 clusters fall into V, R, G and M groups.

The memory footprint varies significantly across the different clusters, e.g., \(\sim 5\text{MB}\) for R0, V0 and X0, while \(\geq 40\text{MBs}\) for G0, V6, X\{7,8,9,10\}. While the higher overhead might indicate a higher incentive to understand and optimize the JS for clusters with larger size, such clusters do not show presence across a wide range of websites and, consequently, their impact is not generalizable. In Figure 6.19, we plot the percentage\(^\text{18}\)\footnote{If the main source of overhead is a general-purpose allocator, e.g., malloc, multiple components may be responsible due to the lack of one-to-mapping between the general-purpose allocator and browser components} of websites with the capsule across websites.

\(^{18}\)If the main source of overhead is a general-purpose allocator, e.g., malloc, multiple components may be responsible due to the lack of one-to-mapping between the general-purpose allocator and browser components

\(^{19}\)Sets are named based on the key component’s purpose during the pageload.
of websites that show a certain cluster and observe that the relatively large clusters (e.g., G0, V6, X\{7,8,9,10\}) are only observed for less than 2.1% of websites. Further, a website may have tens of capsules of one type, while few or none of the other types and, consequently, the total memory overhead of a certain cluster may be amplified (or diminished) based on this factor. We show this behavior in Figure 6.20 where we compare a capsule’s individual memory size versus its cumulative size in a website. Each data point represents a cluster and the sizes are averaged across all websites. Clusters with smaller individual size (y-axis), especially R and V clusters, lead to a much higher cumulative size, since they are called multiple times within a website.

A key consequence of these two trends is that the memory overhead of different clusters is expected to vary across different types of websites, based on their website-specificity and cumulative impact. Among all the clusters, M, G and most X clusters show a more website-specific nature.

Next, we discuss the different cluster types to understand their purpose, their event composition and the source of memory overhead. To guide the analysis, we use Table 6.3 that summarizes the root-cause events for each cluster as well as the percentage of capsules that belong to a certain cluster in our data-set.

### 6.7.2 V clusters

For the V clusters, the v8 allocator dominates the memory overhead, whereas two other allocators (malloc and partition_alloc) only show minor contributions. Such allocation pattern indicates that V capsules mostly concern purely JS activity contained within V8. While their event composition supports this observation (Table 6.3), with JS-events (FunctionCall, v8.compile, parseOnBackground) identified as the root-cause events for all V clusters, some clusters also show non-V8 events, primarily ParseHTML and networking events. Since developers may include JS directly in the HTML through source tags, capsules in V0, V1 and V2 clusters include JS within HTML and the browser emits ParseHTML while processing such JS code. For the latter case, the presence of networking activity is due to the JS code making network requests to either fetch code files (e.g., .js, .html) or media objects (e.g., images) through JS networking APIs, leading to minor malloc and partition_alloc allocations to store objects in native memory (the combined overhead of these allocators is 8MBs at max and, thus, only minor as compared to v8).

Since the overhead primarily stems from the v8 allocator, we leveraged the V8 heap allocation metadata to dissect the allocations. Figure 6.21 presents the source breakdown as the percentage of total capsule
memory, and we observe three key sources: (i) code size that impacts parsing/compilation overhead, (ii) JS and native object allocations, and (iii) “other” category. At the median, 26.5% of memory is attributed to parsing/compiling code, whereas 27.7% is attributed to objects such as strings and arrays. The rest (45.8%) is attributed to “other” category where the profiler is unable to associate small allocations with a pre-defined category and dumps the allocations together, leading to their possibly high aggregate value [220]. Leveraging the JS classification from § 6.5.1, we inspected the capsules with high “other” overhead and noticed that such functions involved primarily computational work, such as regex matching, image modifications and activity trackers, that use heap for storing custom objects.

### 6.7.3 R clusters

Capsules in R clusters heavily rely on cc allocator (i.e, the compositor) for the memory allocations (Figure 6.18). The cumulative overhead of the cc and v8 allocators stands at more than 81% of the total memory and is dominated by cc, while a minor amount (6.3% on average) is contributed by v8. Compositor is responsible for rendering the DOM and graphical elements on the screen, including the layout, images, and animations, etc, and composite all the individual elements of a web page into a single image representation that is displayed to the user. The compositor primarily consumes memory to (i) store the intermediate results of its rendering process, i.e., the changes made by JS APIs to DOM or the webpage objects or structure (called render target memory), and to (ii) store textures to store the graphical elements (such as text, images, DOM objects, etc) that it composites into the final frame (called textured memory).
R’s allocation behavior, as well as events from Table 6.3, indicate that these capsules comprise JS functions that trigger cascading render events. In contrast to the V clusters where the events are largely limited to the V8 component, R’s events span multiple browser components and correspond to a complete workflow of a JS function’s interactions with DOM and the resulting rendering (specifically compositor) work to materialize the changes on screen: JS events (FunctionCall, v8.compile) represent the execution of a JS function that modifies DOM, triggering changes in layout (InvalidateLayout, ScheduleStyleRecalculation, Layout, UpdateLayer, UpdateLayoutTree events). Once the layout changes are translated to the render tree, the compositor commits the final frames (CompositeLayers, FrameCommittedInBrowser events) and finally paints them on screen (Paint). While the memory overhead for JS events is contributed by the v8 (represented by v8’s share in Figure 6.18), the memory overhead of layout and paint contributes to render target and texture memory, respectively, using the cc allocator\(^{20}\). Besides these common events, some clusters also showed other non-compositor events that led to minor non V8/compositor overhead (4.6% in R4 for gpu, and 11.9% and 14.3% in R3 and R4 for general-purpose allocators). We further discuss such events in § 6.10.3.

Since compositor, the key source of R clusters memory, primarily renders web content, we observe its memory overhead to increase with the complexity of content, e.g., size of content in DOM and types of media objects, as also reported by other studies [416, 285]. Consequently, content-rich websites, like sports, entertainment and news, tend to pay a higher R memory cost (47.5%, 56.6% and 38.6% of total JS memory, respectively). However, analyzing the R capsules, we observe a peculiar trend: content complexity is not the only key factor and, in fact, websites with similar content type and size may exhibit different compositor overhead based on their interactions with DOM. To illustrate, let us consider two news websites, bbc.com and dawn.com that have a similar number of DOM nodes (873 vs 826 nodes) and content size (4.3MB vs 4.1MB) added by the R capsules. However, dawn.com incurs a 1.7X higher R memory overhead and the metadata of its R events, specifically layout, show a sharp contrast: median layout for dawn.com results in cascading changes to 2.3X higher number of nodes\(^{21}\).

The explanation for this contrast lies in the complexity of DOM tree structure. As DOM defines the parent/child dependencies, the cascading effects of the JS-DOM interaction with a node (blast radius) increases with the its number of dependent nodes. Higher the number of dependents of a node, the wider is the blast radius of layout changes, and, in fact, the number of nodes affected in a layout (i.e., dirtyObjects metadata) correlates highly with the R memory, with $\rho=0.83$ (p-value < 0.000). Our analysis of the increased overhead and discussion of data with Chrome developers identified that the overhead increases with layout complexity due to amount of intermediary texture representation required to buffer. A similar behavior was also observed with R capsules that involved timers/intervals and frequently updated DOM over time (case-study discussed in next section). R memory’s relationship with DOM/layout complexity is further discussed and validated in § 6.10.4. A key significance of this result is that developer also need to look beyond the JS into the DOM complexity of their websites to optimize the R memory overhead.

\(^{20}\)R4 cluster also show small contributions (8.3%) from blink rendering engine.

\(^{21}\)Calculated using the elementCount and dirtyObjects metadata fields.
6.7.4 G cluster

Among all the clusters, G cluster show the smallest presence (1.3% of websites, 0.07% of the total capsules) and their memory is primarily contributed by the gpu and cc allocators. The GPU component executes graphics and visual effects tasks and Chromium architecture include multiple back-ends for graphical tasks, among which the two most commonly used are: (i) the hardware-based GPU process, and (ii) the software-based Skia library that is part of the compositor [494]. The choice of back-end depends on the type of APIs used by a developer for the animation tasks. Specifically, WebGL-based libraries use the GPU process, while others like Canvas [494] use Skia. Among all the capsules, the G cluster is only observed when WebGL-based libraries, such as create.js [118], are used to create fluid animations, further validated in § 6.10.5.

Analyzing the key events, G cluster comprises a wide range of events, ranging from rendering, animations, GPU to images (Table 6.3). This series of events exist as textures produced by the GPU are directly used by the compositor for rendering the final page. In fact, some of these events are also present in other clusters, such as R1, R4 and X9, and yet no significant gpu allocations are observed. The key reason for this behavior is the selection of animation back-end, where GPU process is only used when WebGL-based APIs are used in the capsules. To evaluate, we inspected the browser process (indicated by the pid in metadata) that called the GPUTask events and found that the Renderer process calls such events for the non-G clusters, indicating the usage of software back-ends.

For the set of websites where G show presence, their memory overhead is significant, ranging between 35-63%. We attribute the key reason for this overhead to bitmap storage (identified by the GpuMemoryBuffer (used_bytes field) [163]). Chrome by default use one-copy texture uploads where GPU maintains a copy of bitmap, serving the basis for animations/visual graphics, and operates on the bitmap to upload the final textures (identified by the texture memory for cc) to the Renderer, leading to cc allocations for the G cluster. We also observe the size of buffers to directly correlate with the G memory ($\rho=0.93$, p-value < 0.001) and their total size makes up ~89% of G memory on median.

6.7.5 M clusters

For the M clusters, we observe that the memory is overwhelmingly contributed by general-purpose allocators malloc and partition_alloc. In contrast to the earlier discussed clusters, these allocators do not map 1-to-1 to a browser component and are frequently used across different Chrome components.

Event composition of M clusters (Table 6.3) show media-specific activity (image and media tasks). While M’s general-purpose allocators are also used for a variety of other reasons (such as storing code files in V clusters), the key browser events for both M clusters concern media, either images or video, content. For a fetched object, the browser has the option of either storing it in cache (using web_cache allocator) or in general-purpose memory. Chrome documentation indicates that large media objects (such as videos) are stored through malloc/partition_alloc as they are not expected to be used across user sessions. Consequently, such objects are not stored in cache to maintain free space for other cache-able objects.

M clusters only show significant memory overhead for streaming websites (31.7% of total JS memory), while it is much less significant for other categories (≤8%). In fact, their overhead directly correlates with
the size of media content and, besides the streaming websites, the highest overhead is observed for news and sports websites with multiple banner images and photo-sharing websites such as tumblr and pinterest. In § 6.10.7, we further show the memory overhead’s relationship with media format.

6.7.6 X clusters

In contrast to other clusters, capsules in X clusters show a mixed behavior where a number of different allocators, and by extension browser components, are involved without a clear trend. In fact, most X clusters involve strong contributions from general-purpose malloc, partition Alloc and shared memory allocators, and a lack of one-to-one mapping of such allocators with browser components makes X clusters the most challenging to explain.

Inspecting the X capsules through our JS classification framework, we observed a key property for their JS: they comprise long functions, either custom functions written by developers or imported from 3rd-party libraries, especially advertisements, trackers or stand-alone web components such as widgets. Developers typically use these functions to offload a series of activity to library calls, e.g., traverse DOM to add event listeners for trackers, infer user preferences to fetch and display user-specific ads, etc., and consequently, their activity spans multiple components such as networking, media storage and DOM interactions. Note that, due to their domain-specific purpose, most X clusters only show scarce presence: X{7,8,9,10} in ≤2.1% of websites, X{3,4,5,6} in 5.6-9.1% of websites (Figure 6.19). Similarly, X3-10 cumulatively make up 4.56% of the capsules in our data-set (Table 6.3). For the X clusters with the highest presence, we observe them to exclusively belong to ad libraries: X0 (51.4% of websites, 14.5% capsules), X1 (30.2% of websites, 4.57% capsules), X2 (29.9% of websites, 4.38% capsules).

The allocation pattern and event composition of some of the X clusters show similarity with other non-X clusters, e.g., X9 events match R clusters. We observe X9 capsules to dynamically add pre-made interactive elements, e.g., widgets, banners, etc., to the main DOM leading to a similar activity as the R clusters (high compositor overhead). In contrast, for some other X clusters that involve layer/frame/paint events, such as X{3,6,7,8,10}, no significant compositor overhead is observed and blink show comparatively stronger allocations. These clusters are primarily observed for ads JS where developers define iframe for ads in HTML, an area with pre-determined dimensions that is later filled with ad content, and the iframes attach a separate DOM to the website’s main DOM. A consequence of this design is that any changes to iframes are contained within the iframe’s DOM and the layout of main DOM remains unchanged. Chrome documentation suggests that Blink renderer is used to render updates that do not involve changes to the layout or geometry of the page and use either blink or partition Alloc for memory [132]. We suspect the overhead of such clusters to emerge due to these reasons. Further, events for some of X clusters show key distinctions: X6 cluster involves capsules that are primarily image focused, while X3 and X10 involved usage of timers to periodically re-paint the webpage.

The long nature of functions and the usage of general-purpose allocators makes it challenging to explain the memory of X clusters through our technique. To gain more insights into such functions, we recommend breaking up the code into smaller functions that can better fit the needs of our approach.
6.7.7 Takeaways

While the nature of JS code may vary across websites, a limited number of memory allocation patterns are observed for the diverse capsules, enabling us to narrow the memory usage trends for the plethora of capsules to a handful of clusters. The clusters provide a framework to decompose a website’s JS memory into a set of building blocks, providing visibility into their generalizability, size of memory overhead and the key factors that impact the memory. More importantly, our measurements unveil the potential optimization targets to ease mobile web’s memory plights.

Memory for V and R clusters, making up the highest number of capsules in our data-set, is a function of coding practices and website DOM design, thus requiring either (i) rethinking of website design to reduce heap wastage, simplify DOM or reduce layout complexity, or (ii) usage of over-the-counter or design of new optimizations to reduce the V8 and compositor overhead on the client-side (e.g., at the expense of lower client-side interactivity by offloading costly operations to the proxy/server-side). On the other hand, for the G and M clusters, the overhead is due to usage of fluid graphics that require the GPU graphics back-end or the quality of media content, respectively: two cases where the developer makes explicit choices to include higher utility content at the expense of memory. Further, while G capsules can be optimized by using a different graphics back-end, the scope of optimizations for the M capsules rest outside of optimizing JS, but optimizing the nature of media content (either size or format) to be more memory friendly. Finally, for the X capsules, the overhead is largely a function of JS for advertisements, trackers or stand-alone web components, e.g., widgets. From a user-perspective, the nature of such components are generally peripheral to the main utility of a websites (e.g., headlines in a news website have a higher utility than ads) and developers can make conscious decisions to reduce their overhead by selectively disabling them for memory-constrained devices. The stand-alone nature of such components makes it especially viable to remove them without disturbing the overall webpage.

6.8 Optimizations

Over the past few decades, a number of JS and web optimizations have been proposed to improve JS efficiency during the various stages in its life-cycle discussed in § 6.2.3. In this section, we evaluate several such optimizations, and leverage JS-Capsules to quantify and analyze their memory impact.
<table>
<thead>
<tr>
<th>Optimization</th>
<th>Scope</th>
<th>Purpose</th>
<th>Potential capsules</th>
<th>target capsules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy parsing [413, 394]</td>
<td>V8 + JS pre-processing</td>
<td>Improved parsing and compilation through pre-parsing.</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Caching [112, 204]</td>
<td>V8</td>
<td>JS computations cached and used across pageloads.</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Lazy render [224, 418, 419, 420]</td>
<td>V8 + Browser</td>
<td>Layout operations are limited to ATP content.</td>
<td>R, X</td>
<td></td>
</tr>
<tr>
<td>Zero-copy rasterize [162]</td>
<td>Browser</td>
<td>Copy tiles across frame to reduce rendering overhead.</td>
<td>R, G</td>
<td></td>
</tr>
<tr>
<td>Proxy-offload [349]</td>
<td>Web infra</td>
<td>JS and DOM operations are offloaded to a proxy.</td>
<td>V, R, X, G, M</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.4:** Optimizations.

Our approach towards selection of a target optimization rests on the following principles.

- First, to ensure that our selections are readily used in practice as a cure to JS inefficiencies, we aim to select optimizations that are among the top recommendations made by the browser/V8 developers.

- Second, to ensure complete coverage over the various sources of memory overheads, the selected optimizations should impact one or more of the memory overhead sources discussed in the previous sections and cumulatively have a complete coverage over the various stages of a JS function’s life-cycle.

- Third, since evaluating an optimization requires practical changes to the test-bed — in form of modifications to JS, V8 or browser behavior — the optimization should be open-source, either as a code-package (e.g., JS framework, modified browser version) or a stand-alone deployment that we can directly incorporate in the test-bed. Otherwise, we face a practical integration challenge where manual re-engineering effort is required to develop the optimization from scratch, often individually for each website, and prohibits the automated applicability for a wide range of websites.

In Table 6.4, we list the set of selected optimizations and Figure 6.22 presents a high-level overview of the different stages in a function’s life-cycle and maps the optimizations’ scope to the different stages. Our selected set of optimizations span from pre-processing JS code, tuning V8/browser behavior to modify parsing/render/paint behavior, to the use of specialized infrastructures to offload or minimize JS overhead on the client-side, and thus provides a full coverage over the various stages of a JS function’s life-cycle. These optimizations are among the top recommendations made by the website and browser developers to improve JS performance and resource usage [147, 397, 398, 250], and are expected to impact one or more types of capsules, as listed in Table 6.4 (potential target capsules). Note that, our list excludes special-purpose JS frameworks such as React and Angular and closed-source JS optimizations proposed in academia [204, 521, 379, 376, 280]. These optimizations do not meet the 3rd principle criteria, as the former requires a complete overhaul of the website code to replace JS APIs with their specialized APIs, thus requiring manual re-engineering effort for each website, and, for the latter, no open-source implementations are available that we can integrate in our test-bed. However, to circumvent these practicality challenges, we took two steps: first, we searched for open-source optimizations that follow the same motivation and
added them to our testbed (e.g., JS caching for [204], proxy-offload for [521, 379]), and second, we designed custom optimizations that aim to mimic the key principles followed by JS frameworks such as React. However, we acknowledge that the approximate replacements may not always be fully representative, e.g., our proxy-offload replacement lack the sophisticated dependency resolution proposed in [521, 379, 376].

Next, we evaluate the selected optimizations and leverage JS-Capsules to quantify and analyze their impact on memory. The measurements are performed for 50 webpages, randomly selected from the Alexa top 1K list. We implemented each of the optimizations in our test-bed and generated JS-Capsules with and without the optimization enabled, in order to evaluate and understand its impact\(^{22}\). In Figures 6.23 and 6.24, we present the percentage JS memory saving as well as the cluster level contribution for the different capsules. In § 6.10.8, we further evaluate if the selection of random websites introduce any bias into our evaluation results and, comparing the capsule distributions for the 50 websites with the Alexa top 1K, report that the sampled websites show a fairly similar capsule distributions.

6.8.1 Dead-code elimination

While analyzing the V capsules, we observed that capsules with high memory tend to include functions with comparatively large code size, i.e., number of bytes, due to the parsing/compilation overhead of the larger number of bytes. Developers often import large libraries where all the included functions are not necessarily required and we show this trend in Figure 6.25 where we plot the percentage of unused code bytes for functions in a capsule: 51% of bytes of code are unused at median (78% at tail). Removing unused code, as proposed by optimizations [291, 168, 528, 75, 179], is expected to reduce V8 parsing/compiling overhead and we tested Muzeel [291], a cutting-edge technique for inferring unused code by dynamic interactions with the webpages, to understand the impact.

For most websites, Muzeel’s impact on memory is modest (~2.3% at median in Figure 6.23) and only a small proportion (18%) of websites show higher than 10MBs reduction. A key reason for the modest savings is that websites today employ JS pre-parsing optimizations where functions that are unlikely to be used are only partially parsed. Consequently, the memory overhead of parsing, compiling and building state, e.g., allocating objects, is minimized if a function is redundant. In fact, we observe that for the websites that show

\(^{22}\)For OperaMini, we do not generate the capsules as the JS is completely offloaded to a proxy and not executed at the client-side.
the highest savings with Muzeel (28% at tail in Figure 6.23), a similar memory impact can be achieved by tuning the pre-parsing behavior as discussed in the next subsection.

Comparing the capsules for the default vs Muzeel-ed webpages, the source of memory savings are highly inclined towards V capsules, while some websites also showed reduced overhead for M capsules (≤5% at tail) due to removal of redundant media pre-fetches (Figure 6.24). Since the removed functions are redundant, their overhead is overwhelmingly limited to parse/compile and, consequently, their impact is largely limited to the V memory.

Some other works [99, 75] aim to further reduce the code size by removing functions such as from advertisements and trackers, with the assumption that such code holds low-utility from a user-perspective. Our case-study of such functions indicates that such optimizations are expected to save more memory (e.g., between 17-31% for content-rich websites), however at the cost of removing functionality that might be crucial for retailers (such as ads and interest tracking). Due to removal of ads/trackers, these optimizations directly reduce overhead of X capsules.

### 6.8.2 Lazy parsing

To reduce parsing/compiling overhead, V8 engine by-default lazy parses all the functions, where the bare minimum work required to verify syntax and compile a functions is done in a pre-parse step [413]. When a pre-parsed function is called later, it is fully parsed and compiled on-demand. Consequently, in case of functions that are immediately invoked, there is often a performance trade-off as extra amount of work (i.e., redundant pre-parse, followed by a full-parse) is performed. The pre-parse behavior can be tuned by adding “hints” [394], and we leveraged a popular framework, optimize.js [394], to the the memory impact for three scenarios: (i) default where code is kept in its original state, (ii) full-parse where all functions are fully parsed, and (iii) selective-parse where the top 25% functions (based on their memory overhead) are fully parsed. Figure 6.26 plots number of functions eagerly-parsed in the 3 cases. While the trend varies across websites, thousands of functions are marked for eager full-parse in default, and we observe default and selective-parse to follow a similar distribution.

Intuitively, one would expect full-parse to lead to the highest memory overhead, as all functions are fully parsed/compiled. Similarly, default should lead to the lowest overhead, assuming that the developers are correctly using the pre-parse optimization. In Figure 6.23, we plot the difference in memory for full-parse versus default/selective-parse, with a positive value representing the amount of memory saved as compared to the full-parse case. For a vast majority of websites (i.e., ~80%), the different parsing behaviors led to an insignificant difference in memory (less than 3MB). For these websites, we observed that 87% of the selective-parse functions are already marked for eager full-parse in default, indicating that developers are correctly employing the optimization. Moreover, the impact was limited to capsules that involved V8 activity, i.e., all V and a subset of X capsules (Figure 6.24), since parsing and compiling operations are contained inside V8.

The lower tail for default and selective-parse in Figure 6.23 shows a counter-intuitive behavior where

---

23 The top 25% functions amount to 79% of the total JS memory at median and are identified in a prior step.
eagerly parsing all functions lead to a lower memory usage (upto 20MBs for surveymonkey.com). Analyzing the capsules, we observed two key reasons for this trend: (i) only a small proportion (7% at max) of the functions in these websites are unused functions and therefore the overhead of pre-parsing is redundant, and (ii) fully parsing all the functions in the first place allowed the V8 engine to improve the garbage collection, as it had gained complete information about the variable/object allocation and dependencies. Taking surveymonkey.com as an example, we had identified earlier (in case-study section) that the website shows a peculiar V8 memory trend where 24.3MBs are allocated for strings that are unused. Comparing the full-parse case with default, these string allocations were pre-emptively garbage collected for the former case, leading to improved V8 memory. We do not observe similar garbage collection events for the default case and the V8 is required to hold the allocations, even though the string objects are redundant.

6.8.3 V8 caching

When a function is called, V8 can either compile and execute the function from scratch or, if the V8 and browser flags are correctly configured, reuse the cached state for the function [112]. The cached state is stored on disk or V8 heap and can have two memory consequences: (i) V8 relies on developers for not changing the file URLs and not modifying the execution behavior to maximize the cache hit rate, and if these directions are not followed, memory space is wasted for caching state that is unlikely to be used, and (ii) since V8 interactions with the DOM are a key cause of memory overhead, caching that results in reducing this overhead is expected to significantly save memory, especially for rendering.

Comparing the capsules with and without caching, we did not observe any significant change in the memory allocations (Figure 6.23) and we identify three key reasons: (i) Websites do not produce a significant amount of cached data, as we show in Figure 6.27, even at tail, the total size is less than 10MBs. (ii) We observed high hit-rates for the cache, with 93% of cache entries for a website hit at median (73% at p5, 100% at p95) and, consequently, no significant cache space is wasted. (iii) Finally, the functions that were cached did not involve any interactions with DOM and, consequently, the memory overhead associated with rendering stays the same. Although some research works have proposed more effective ways for caching JS computations [204], we were unable to test due to unavailability of open-source implementations.
6.8.4 Batch JS timers

Developers often use JS APIs such as `setTimeout` and `setInterval` for scheduling callbacks in the future, e.g., such APIs are used to constantly update counters on `worldometers.info`, or add animation in `drupal.org`. Such timers are expensive in terms of CPU [488] and, in fact, our measurements in earlier sections indicate that memory overhead increases with the timer frequency due to excessive interactions with DOM and consequent paints. By default, Chromium uses several heuristics to reduce their overhead, such as batching the timers and limiting their frequency once per minute if the tab is hidden [488]. Although such optimizations have significant impact (upto the 28% reduced battery usage [489]), they are only limited to background pages, i.e., pages hidden for more than 5 minutes [488].

We tested the impact of batching such events for foreground pages by compiling Chrome with flag `throttle-foreground-timers` [486, 487] set to enabled. This optimization runs DOM timers with a non-zero delay on a regular 30 Hz tick, instead of as soon as their delay has passed [486, 487], roughly translating to a 32ms delay [490]. Comparing the JS memory, we report only a minor change in overhead for most websites (Figure 6.23), however, up to 13.2% savings are observed for the tail websites. Analyzing the capsules for the cases when the overhead is reduced, we observe that the tail websites include R and X clusters, primarily X3 and X10, where timer-based activity (e.g., `RequestAnimationFrame`) are among the root-cause events. Due to batching, the reduced rate DOM leads to a reduced compositor activity to update and store the intermediary
textures, required for generating the new frames.

6.8.5 Lazy render

Chromium, by default, generates the render representation for the entire webpage and, based on the user’s current viewport, paints the corresponding content on screen [146, 224]. Frameworks, such as React, allow developers to optimize rendering by lazy loading webpage components until they are required [418, 419, 420]. Similarly, Chrome introduced a loading attribute in 2019 through which images can be lazily loaded [224]. Since mobile webpages use a portrait orientation and tend to be vertically long, and lazily loading objects, documents or parts of the webpage optimizes resource usage as the respective part of the page are only rendered when required (e.g., user scrolls to the bottom of the webpage).

While the Chrome optimization is straight-forward to implement (i.e., by setting the loading attribute), adding lazy loading to an arbitrary website requires re-engineering its code and design to use specific libraries, such as React. To circumvent this practical challenge and test the impact of lazy loading for an arbitrary website, we implemented a JS framework to control browser rendering behavior. The framework uses MutationObserver for controlling the render behavior to emulate a scenario where the browser only renders the above-the-fold content. At a high-level, it inspects the position of a visible DOM node on screen and disables rendering for the nodes that are outside above-the-fold. To disable rendering, we set the display:none CSS property for the respective node that forces the browser to skip the node during render tree construction and the node plays no part during the rendering phase. We validate the correctness of this optimization by inspecting the browser events are ensuring the absence of render-events for the disabled nodes.

We observe moderate to significant savings for lazy render: 9.3% and 31.8% at median and tail, respectively (Figure 6.23), with R capsules being the key source of savings (7.1% and 24.5% at median and tail in Figure 6.24), followed by X and M capsules (5.4% and 2.9% at tail, respectively). Dissecting the savings, we observe the reduction in extent of layouts to be the key factor: limiting the render representation to the above-the-fold content leads to a reduction in the render-tree size and, consequently, the nodes outside the current-view are removed from the layouts. As discussed in R cluster memory overhead, this leads to a direct impact on the complexity of layouts and, thus, the compositor overhead. In fact, we measure 93% of R savings to directly emerge from the cc allocator, due to a reduction in render-target memory.

6.8.6 Zero-copy rasterize

While painting tiles on screen, Chromium rasterizes web content onto a bitmap and then uploads the bitmap to a texture, resulting in a CPU to GPU copy, which zero-copy aims to eliminate [163]. While zero-copy rasterize is primarily not a JS optimization, we evaluate its impact for websites where JS resulted in significant GPU memory (i.e., websites with G capsules). To enable this behavior, we compiled Chrome with flag enable-zero-copy [162] set to true.

Comparing with the default case when this feature is disabled, we do not observe any significant change in JS memory (less than 1MB), the key reason being Chrome by default uses one-copy rasterize and, as reported by other works [163], the two features incur similar memory overhead.
6.8.7 Proxy/server Offload

Since client-side rendering is a key contributor to memory, an obvious optimizations direction should be reducing the extent of client-side rendering. A number of optimizations are used today, ranging from Server-Side Rendering (i.e., server renders the DOM at request time and send the rendered DOM to the client), Pre-rendering (i.e., static DOM is generated at application build time), to Pre-rendering with Re-hydration [250, 399, 398]. These optimizations are expected to bring significant memory savings as they do not suffer from the high cost of building DOM through JS, and instead serve a complete or partially built DOM, that is later complemented by client-side JS (i.e., Re-hydration [250]). Our measurements with Opera Mini [349] (offloads page load to a proxy) shows up to 33-35% improvement at median 24. With Opera Mini, client browsers simply fetch a final, rendered markup document from a proxy server, which performs the heavy lifting to load the page. However, these optimizations often lead to privacy concerns [204] and reduced interactivity, as client-side JS is completely or partially eliminated or requires network interactions with the proxies.

6.8.8 Takeaway

Evaluating a diverse range of optimizations, our analysis shows that optimizations that tame the DOM rendering aspects of JS, either by limiting the extent of render-tree or by offloading these interactions to a proxy, result in significant, cross-component memory savings for a diverse range of websites. The potential of V8-based optimizations, such as improved parsing/compilation, caching or batching, are only limited to optimizing V capsules (or V8) memory and does not bring significant savings for other types of capsules. While tons of work is done in developing profiler for V8 [11, 495, 170, 441, 539], the lack of impact on other capsules for most browser-based optimizations indicate that there is a need for holistic profilers for other browser components, such as compositor, for an in-depth understanding of JS APIs interactions with such components. Such profilers can be instrumental in understanding the the impact of JS on browser mechanics in depth and uncover potential opportunities for novel optimizations.

6.9 Conclusion

Although JavaScript memory is a crucial aspect of the mobile web page load process, we lack sufficiently fine-grained and precise tooling or methodology for accurately and effectively capturing JavaScript memory. In this paper, we develop a novel framework, JS-Capsules, that builds on knowledge of common browser architectures and targeted logging information to capture fine-grained and precise JavaScript memory measurements. Using our tool, we analyzed 1000 websites and report on the common patterns and discuss the impact of our findings on several practical optimizations. Our measurements show that JavaScript results in significant indirect memory overheads (i.e., non-V8 memory) and a wide range of browser events, especially rendering, contribute to the overhead.

24We only report the overall JS memory savings as JS is not executed at the client-side, thereby prohibiting us from generating capsules. In this case, we calculate the JS memory through the ground-truth JS memory measurement strategy discussed in § 6.6.
While significant amount of work has been undertaken to develop profilers for V8, our measurements indicate that there is a need for holistic profilers for other browser components, such as compositor, that provide an in-depth understanding of JavaScript APIs interactions with such components. Most importantly, our measurements also highlight a need for fine-grained models, benchmarks, and statistical techniques to adequately reason about the finer level of granularity that our tool provides.

6.10 Supplementary Results

6.10.1 Event metadata correlation

In Table 6.5 and 6.6 (split into two due to large vertical size), we plot the correlation coefficient for all possible metadata fields for the key events listed in Table 6.3. To calculate the coefficients, for each cluster group type, we generate timelines for the capsule memory and event counters, correlate the two time-series. There’s a few reasons why correlation would not yield any results (i.e. NaN value): (i) the metadata is a string (e.g., filename or URL) and hence has no quantitative value, (ii) the p-value results in higher than 0.05, or (iii) the respective event/metadata is not a key event for a cluster and does not show-up for its capsules.

6.10.2 Event-based clustering

A potential option is to classify the capsules on the basis of their browser activity, the key intuition being, as different capsules may emit similar cascading events, we can leverage the events for clustering and simplify the analysis by focusing on the individual clusters, instead of individual functions or capsules. For this purpose, we represented each capsule as a frequency histogram of browser events and experimented with a number of clustering techniques including Kmeans, SVM and hierarchical clustering, and empirically evaluated Kmeans to be the best option. However, we noticed two issues with a straight-forward event-based clustering. First, taking a purely event-centric approach assumes that all constituent events in a capsule are equally responsible for memory. In fact, although the event makeup for several capsules belonging to the same cluster were similar, their allocation patterns varied widely, leading to either significant variance in memory for capsules within a cluster, or use of completely different allocators, and by extension, browser components for contributing the memory. This indicated that the clustering should take into account the different memory allocator make-up of capsules to account for the memory impact of the constituent events. Second, we observed a bias issue where the number of times an event is called showed significant divergence. Although data pre-processing such as filtering and normalization can help, we observed the bias-ness to persist. We also tried increasing the number of clusters so that the less-frequent events’ importance is preserved, but that defeated the purpose of clustering as it yielded a large number of clusters (~250).

6.10.3 R clusters non V8/compositor events

Apart from the 10 events that are common across all R clusters, some R clusters also involve specific events:
<table>
<thead>
<tr>
<th>Event name</th>
<th>Metadata field</th>
<th>Data type</th>
<th>R</th>
<th>G</th>
<th>M</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionCall</td>
<td>columnNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>functionName</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>timeNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>scriptId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>0.19</td>
<td>0.09</td>
<td>0.07</td>
<td>0.17</td>
</tr>
<tr>
<td>v8.compile</td>
<td>columnNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>lineNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>notStreamedReason</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>streamed</td>
<td>bool</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>fileName</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>0.83</td>
<td>NaN</td>
<td>NaN</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>partialLayout</td>
<td>bool</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>totalObjects</td>
<td>numeric</td>
<td>0.51</td>
<td>0.23</td>
<td>0.23</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>layerRoots</td>
<td>array</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>depth</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>nodeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>quads</td>
<td>array</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>elementCount</td>
<td>numeric</td>
<td>0.57</td>
<td>NaN</td>
<td>NaN</td>
<td>0.31</td>
</tr>
<tr>
<td>Layout</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>nodeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>columnNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>functionName</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>timeNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>scriptId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>0.11</td>
<td>0.01</td>
<td>0.03</td>
<td>0.09</td>
</tr>
<tr>
<td>InvalidateLayout</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>nodeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>columnNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>functionName</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>timeNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>scriptId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>0.11</td>
<td>0.01</td>
<td>0.03</td>
<td>0.09</td>
</tr>
<tr>
<td>ScheduleStyleRecalculation</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>nodeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>layerTreeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>frameSeqId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>layerTreeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>UpdateLayer</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>elementCount</td>
<td>numeric</td>
<td>0.51</td>
<td>0.23</td>
<td>0.23</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>clip</td>
<td>array</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>layerId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>nodeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>createFrame</td>
<td>array</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>processId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>CompositeLayers</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>layerTreeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>UpdateLayoutTree</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>elementCount</td>
<td>numeric</td>
<td>0.51</td>
<td>0.23</td>
<td>0.23</td>
<td>NaN</td>
</tr>
<tr>
<td>Paint</td>
<td>clip</td>
<td>array</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>layerId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>nodeId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>createFrame</td>
<td>array</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>processId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>CompositorFrameSink</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>RequestAnimationFrame</td>
<td>id</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>columnNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>functionName</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>lineNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>scriptId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>0.11</td>
<td>0.01</td>
<td>0.03</td>
<td>0.09</td>
</tr>
<tr>
<td>XHRReadyStateChange</td>
<td>frame</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>readyState</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>columnNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>functionName</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>lineNumber</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>scriptId</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>objectLsize</td>
<td>numeric</td>
<td>0.35</td>
<td>NaN</td>
<td>NaN</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Table 6.5: Correlation for key event metadata.

- R1 involves `requestAnimationFrame()` API that is primarily used for updating animation render [527]. Such capsules use JS to add dynamic animations that are updated frequently over time, or wait for
<table>
<thead>
<tr>
<th>Event name</th>
<th>Metadata field</th>
<th>Data type</th>
<th>R</th>
<th>G</th>
<th>M</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUTask</td>
<td>renderer_pid</td>
<td>numeric</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>used_bytes</td>
<td>numeric</td>
<td>0.35</td>
<td>0.93</td>
<td>NaN</td>
<td>0.31</td>
</tr>
<tr>
<td>DecodeImage</td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>object_size</td>
<td>numeric</td>
<td>0.21</td>
<td>0.61</td>
<td>0.67</td>
<td>0.32</td>
</tr>
<tr>
<td>DecodeLazyPixelRef</td>
<td>pixelRefId</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>ImageDecodeTask</td>
<td>pixelRefId</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>ImageUploadTask</td>
<td>pixelRefId</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>media/base</td>
<td>N/A</td>
<td>N/A</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>media/filters</td>
<td>N/A</td>
<td>N/A</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Animation</td>
<td>N/A</td>
<td>N/A</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>SendRequest</td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>ReceiveResponse</td>
<td>url</td>
<td>string</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>object_size</td>
<td>numeric</td>
<td>0.34</td>
<td>0.49</td>
<td>0.49</td>
<td>0.45</td>
</tr>
</tbody>
</table>

Table 6.6: Correlation for key event metadata (continued).

![Figure 6.28: R memory impact of DOM complexity.](image)

user-initiated interactions for animation updates. These capsules also involved the usage of timeout/interval functions and we further discuss the memory overheads of timers while discussing the related optimizations in § 6.8.

- R3 and R4 involve XHRReadyStateChange that corresponds to completion of an AJAX network request and trigger of callbacks. While no significant network memory overhead is incurred, we observe higher contribution for storage allocators for these capsules to store the fetched media content (11.9% of memory for R3, 14.3% for R4).

- R4 involves GPUTask event and, among all the R clusters, shows the highest gpu memory overhead (4.6% of memory). Such capsules involve executing of libraries to create dynamic animations and we further discuss the memory overhead of GPU in the discussion of G clusters.

6.10.4 R clusters validation

Since the bulk of R memory originates from events that concern rendering DOM, we inspected the DOM structures and observed that websites with the highest R overhead tend to comprise the heaviest DOM trees: ~2.2k-3.1k nodes at median (4.2k-7.3k at tail), with depth ranging from 15-44. Further, such websites heavily rely on client-side JS to supplement the base DOM structure defined in the HTML, with up to 70% of DOM content and nodes dynamically added on run-time through JS APIs. In contrast, websites with low R overhead comprise ~4X smaller DOMs on average, have a relatively flat structure (2-5 depth) and use
client-side JS to a similar extent (62% of DOM added by JS on average). In fact, the size of content added by JS correlates moderate-to-strong with the capsule’s R memory (Spearman’s coefficient [362] of $\rho=0.79$ with p-value \( \leq 0.000 \)). The correlation indicates that JS memory of a website is expected to increase based on JS contribution’s on populating the DOM content. However, we also observed a sharp contrast in R memory of capsules that led to different extent of cascading changes in DOM.

As DOM defines the parent/child dependencies, intuitively, the cascading effects of a JS function interaction with DOM should depend on the DOM’s complexity, e.g., changing the layout of a parent node cascades into changes to its dependents, if any exists. Inspecting the R capsules, we observe two factors that impact the extent of cascading events: (i) type of DOM tree (e.g., flat) and (ii) the location of node being manipulated; the former dictates the dependency between the nodes, while the latter dictates the blast radius, i.e., extent of changes cascaded to the children nodes.

To validate this observation, we designed synthetic pages\(^{25}\) and use JS-Capsules to capture fine-grained interactions. We test a variety of different dimensions: content types (text, images, videos), different DOM sizes (number of nodes between 50 and 500), different JS API calls and different DOM tree structures. While, as expected, the compositor overhead increased with the content/DOM size, surprisingly, the different DOM structures exhibited widely different overhead for the same content/DOM size/type and JS APIs. We present the results in Figure 6.28 where we plot memory as a function of time\(^{26}\) for four variants with varying blast radii. Low, Medium and High blast radius refer to trees where adding a new node results in re-layout for 10%, 25% and 100% of the nodes in DOM, respectively. Figure 6.28 normalizes the memory timeline of these variants by No blast radius case, i.e., no existing nodes are re-layed out when the node is added. We observe that the memory cost increases with the blast radius due to a higher number of rendering events to re-layout the affected nodes, specifically Layout and CompositorFrameSink, the former calculates the layout of DOM content on screen, while the latter produces frame data that is later painted on the screen. Our analysis of the increased overhead and discussion of data with Chrome developers identified that the overhead increases with layout complexity due to amount of intermediary texture representation required to buffer.

6.10.5 G clusters validation

One of the websites with G clusters was drupal.org, homepage for a popular web content-management platform. The website consumed 143.5MB of JS memory of which 89.7MBs were contributed by G0 (60.2% and 37.8% for gpu and cc allocator, respectively). Inspecting the capsules, the overhead emerged from an animations library (CreateJS) that created dynamic animations and required GPU memory to create the textures and compositor memory to generate the required frames. Interestingly, the webpage included a button to ”stop animations” that disabled the usage of CreateJS. We added a minor modification to disable the animations by default at the start of page load and the overhead for G0 was reduced to 17.2MBs.

---

\(^{25}\)The design of the synthetic pages is rooted in DOM properties measured for real websites and is discussed in § 6.10.10.

\(^{26}\)The timeline starts with the beginning of page load and ends at 30 seconds to accommodate the allocations until page load is complete. The onLoad event is fired at 7s, 12s and 27s for the low, medium and high case, respectively. The 3 web pages show different page load times as the rendering computational work increases with the complexity of DOM trees.
To validate that the top X clusters belong to ad/tracker JS, we leveraged our test-bed to remove the respective JS functions from the website code in a pre-processing step. Loading the websites without the ad/tracker JS led to removal of the $X \{0, 1, 2\}$ capsules and their respective memory overhead.

### 6.10.7 Impact of media formats (M clusters)

Recall, that we observed the presence of M clusters mostly in streaming or media-content heavy website and the memory overhead is attributed to media storage in native memory. For instance, out of a total 81.4MBs of JS memory for *ustream.tv*, a video streaming platform, 37.7MBs to attributed to just M clusters. Although the media object size is expected to be a key factor behind this overhead, interestingly, we observed the media format is also a key factor.

Focusing on the images, the target streaming websites used four image formats, *png, jpeg, svg* and *webp*, with *jpeg* being the most popular. We performed an experiment where, for each image, we created different version of image formats with exactly the same number of pixels and evaluated how the overhead varies for the different formats. At median, we observed webp to incur the lowest overhead followed by png (4.7% higher than webp), jpeg (9.1% higher than webp) and svg (29.3% higher than webp). A key reason why svg incurred highest overhead is that svg format can be dynamically manipulated with JS (e.g., add animations) and generally requires a higher amount of metadata than other formats.

### 6.10.8 Evaluation of bias for random 50 websites

To evaluate if the randomly selected websites introduce any bias in the evaluation of optimizations, we compared the capsule distributions for the 50 random websites with the Alexa top 1K. We generated CDF of a cluster group’s memory contribution and, in Figures 6.29 and 6.30, compared the median and tail contribution for the two sets of websites. To generate these figures, we first calculate memory contributed by a cluster (e.g., V) to the website’s total JS memory and, for each cluster, generate two CDFs: one for the 50 websites while other for the Alexa top 1K websites. Figures 6.29 and 6.30 then plot the difference at median and tail, respectively, for the two CDFs for each cluster.
For the median comparison case (Figure 6.29), we report fairly similar numbers for the two sets of websites: at worst the memory impact of a cluster group showed less than 1.75% difference. For the tail comparison case (Figure 6.30), we observed slightly higher deviation for the R (∼12%) and M (∼10%) clusters. While the difference at tail is large for two of the clusters, the fairly similar distribution at median shows that the random samples, for the most part, do not introduce any significant bias.

6.10.9 AMP memory savings

Accelerated Mobile Pages (AMP) [169], a recent initiative by Google, aims to address CPU, bandwidth, and memory constraints by providing web developers with an alternative ecosystem for developing lightweight pages. In AMP, pages are developed using an alternative and lightweight image format and JS use is constrained. Taken together, the constraints ensure that the webpages are lightweight. To test AMP, we collected top-50 Google Trends for 2020 and searched the keywords to collect 50 distinct webpages with both variants. Comparing the memory usage of AMP vs non-AMP pages, we observe AMP to have significant impact on memory (43% at median).

We observed significant impact of reduced scripting and rendering overhead as the key factor behind the memory overhead (93% of the savings were due to reduced v8 and cc allocators). In AMP, JS is only allowed in constrained environment (e.g., iframes) and developers are required to use AMP APIs, custom developed by AMP developers with resource efficiency in mind, to add functionality to their webpages. These APIs are developed with the web best practices in mind, such as no excessive layout thrashing, limited JS-DOM interactions, small JS code size to reduce V8 overhead, etc.

6.10.10 WebPage structure characterization

Our characterizations provide us with distribution with which we can create synthetic pages. Below we highlight key observations from our analysis:

**DOM structure**: DOM structure defines the parent/child dependencies between DOM objects/nodes (e.g., div, p, img etc.). Though websites can have very diverse DOM tree structures, we observe a few common patterns. At a high level, we discovered three key DOM structures: (i) **flat** where a single parent node fathers all children (representing the tail website and iframe DOM). Iframes (Ads, Tracking) usually have this structure, i.e., one or two nodes parent all the rest of then nodes and most nodes are direct children of the head/body nodes. (ii) **single-child** where nodes are connected in a chain with each node fathering a single node (the extreme case), and (iii) **multi-child** where a node is a direct parent of 2-50 nodes (dense trees with varying width). This pattern is often used by a page’s mainframe, which has mostly complex interactions with up to 44 depth and only a small proportion of nodes directly referenced by the head/body. This complexity leads to a dense-tree structure, where children nodes are distributed across several parent nodes.

**HTML vs JavaScript added content**: Websites tend to vary in their use of JavaScript to add content dynamically. Most simple websites, e.g., wikipedia.org, define their content directly in HTML, whereas more complex websites, e.g., worldometers.info, either create DOM nodes dynamically through JavaScript or use JavaScript to modify HTML-defined nodes.
6.10.11 Generating Synthetic Pages

We generate synthetic pages in two steps, first creating a template and then populating the content. We elaborate on these below:

**Phase 1:** We generate a number of template pages by defining three dimensions: (i) number of DOM trees in the webpage (specifically, one main and N iframe). (ii) the DOM tree structures for each index/iframes. For the DOM tree types, we test three base cases (flat, single-child and multi-child) and further add between 0-5 iframe DOM trees having flat or multi-child structures. Similar to number of nodes, we select the five percentiles from the observed distribution to set the number of children for the multi-child case (i.e., we find the number of children each node must have to fit the N nodes in a tree with depth D). For iframes, we use flat and multi-child trees and build the trees in a similar fashion. (iii) the number of nodes in each tree. For number of nodes, we select five values from the observed distribution: \{5th, 25th, 50th, 75th and 95th\} percentiles.

We create template pages by combining these three dimensions and discard template pages that violates the observed distributions, e.g., we discard any single-child tree with more than 44 nodes because 44 is the max_depth observed.

**Phase 2:** In the second phase, we populate the content nodes in the templates. We selected five tags that are unique with respect to their operations: div, p, img, button (with click event), form and use the top-10 calls for generating content and manipulating DOM/styles. The first phase use divs for defining the trees, and in this phase, we insert content nodes (i.e., tags) into divs to reach the total number of nodes in the tree. For text size (p tags), we generate random text and test five cases: total text in the webpage to match 1kb, 10kb,
100kb, 1mb, and 10mb. For the images, we generated 10 random images with random number of pixels. For each number of pixels, we test four variants: `png`, `jpeg`, `svg`, `webp`. The final sizes of images are between 10kb and 1mb.
Chapter 7

Discussion

In this chapter, we discuss the applicability of CDPlane under constrained environments, such as 3rd party CDNs that do not share the Hyperscaler control properties. We focus the discussion on the following questions: (i) For the three contributions, what are the control requirements over the infrastructure?, (ii) If we relax the control constraints, to what extent can we still improve the QoE?, and (iii) Is there a future where such mechanisms can be applied to a connection that spans multiple organizations?

7.1 What are the control requirements?

For the three components proposed in this dissertation, the control requirements widely vary.

JS Capsules, being a client-side memory analytics technique, does not require any control except the client browser layers. We primarily focus on the use-case where a developer generates the JS Capsules in a local test-bed, analyzes the generated reports to understand the memory usage and selects the appropriate optimization based on the key overhead root-causes. However, the application of the optimizations may require some degree of control over the other layers. In the simplest case, the developer requires control over the website code in order to optimize specific JS functions or website’s content or structural properties. For more complex cases, such as JS offload optimizations, a higher level of control is required over the infrastructure in order to provide the remote computational environment to run the client-side JS.

Configanator requires control over the edge tier (server application, networking stack and OS) and JS layer: control over the OS is required to install the data-path components for capturing connection characteristics and adding tuning flexibility, control over the network stack is required to tune the configurations, and control over the JS layer is required to inject code snippet to capture performance metrics (i.e., page load time). A key advantage of the lack of cross-tier control requirement is that Configanator’s design can be readily applicable to any CDN that retains control over the edge, e.g., 3rd-party commercial CDNs. However, arguably the control over the client-tier can further help in refining the configuration decisions as the

---

1 Modern browsers support “Performance Timings” APIs for easy and accurate measurement of PLT and adding such JS snippet does not necessarily require origin control, e.g., based on the customer contract, CDNs may have the authority to inject such JS at the edge proxy.
client has the most recent view of their network state. Hyperscalers can leverage their client-side control to generate signals [91, 84, 117, 56, 134] from the clients that can be helpful for making the configuration selection decisions, similar to video tuning systems [25, 255].

Among the contributions, Zero Downtime Release requires the highest level of control over the end-to-end infrastructure. As discussed in Chapter 4.4.4 although the three mechanisms proposed for Zero Downtime Release have no inter-dependencies and are used concurrently, they differ in their control requirements due to domain-specific nature of their targeted connections. The control requirements are stated as follows: (i) socket takeover works in the context of a single server machine and the control requirement is only limited to the individual machine’s OS and server/proxy application without any cross-tier requirements, (ii) partial post replay works between server back-ends (HHVM) and proxy servers at the origin and control over both of these layers is required for the request hand-overs, and (iii) downstream connection reuse works across different tiers and requires control over the proxy servers at the origin and the edge, as well as the client-tier.

A consequence of the different control requirements is that the proposed contributions may not be readily applicable to a CDN that lacks control over the various tiers. Among the different tiers, having control over the edge-tier is the most crucial as the edge directly connects with the clients and control over the edge enables the application of dynamic protocol tuning, website optimizations and socket takeover for proxy restarts. However for the rest of the mechanisms, the control over the other tiers is essentially required.

7.2 What if we relax the control requirements?

Among the three types of CDNs discussed in Chapter 2, commercial and meta CDNs do not exercise control over the various tiers and the lack of control may limit the applicability of the mechanisms introduced in the thesis. Since meta CDNs act primarily as a broker and use the connectivity and server deployments provided by the commercial CDNs, we specifically focus the discussion on the control limitations of the commercial CDNs. Relaxing the control requirements, i.e., assuming that a CDN does not have control over some certain web stack layers, is expected to impact the QoE in two ways.

First, the applicability of a mechanism may be either partially or completely prohibited, e.g., socket takeover without the control over QUIC state only prevents disruptions for the TCP traffic, partial post replay and downstream connection reuse are not applicable without the control over the downstream proxy. Similarly, for Configanator, the configuration tuning would be limited to a narrower set of knobs, if the CDN does not control all the edge networking layers. Consequently, the QoE benefits will only be limited to the layers where the mechanisms can be still applied, for instance, in Chapter 4.6, we discuss the scale of disruptions for the different types of services and limited control for QUIC, publish/subscribe or HTTP uploads will lead disruptions for the respective services: QUIC disruptions in order of tens of thousands of connections per instance, ~6.8M terminations at median per web-tier restart for HTTP uploads, ~20% higher CPU overhead

Control over the client-tier for DCR is only required when the user application is the next downstream to a restarting proxy, i.e., edge proxy restarts.

Assuming that the CDN can legally modify website code/content.
Second, as the access may be limited to a subset of connection state or page load layers, the outcome of the decision choices, e.g., selection of protocol configurations, choice of JS optimization, will be based on possibly incomplete set of features and thereby is not expected to be on the same par as with the complete information. For instance, the lack of control over the JS layer may result in lack of access to the PLT information and the control algorithm for the protocol tuning may resort to using other performance metrics, such as time-to-first-byte (TTFB) or time-to-last-byte (TTLB), that are readily accessible from the server application. However, the usage of such performance metrics is not expected to always optimize PLT as metrics like TTFB or TTLB only capture object-level delivery performance, while PLT captures the performance of entire end-to-end page load performance. Similarly, for JS Capsules, the developer will not be able to get a complete picture of her JS’s indirect memory allocations for the various browser layers, if the access is limited.

To further understand the implications of lack of control on the quality of decision choices for Configana
tor, we designed an experiment where we modified the algorithm to learn on a subset of all features, i.e., simulating a CDN that does not have access to all connection/request features. Figure 7.1 plots the PLT improvement for the various cases and we observe that limited visibility into the features can significantly harm the decision quality, e.g., the PLT improvement shows a significant drop (2nd and 3rd boxes) if the “website” feature is not used for configuration prediction. Among the four features, website (\textit{site}), latency (\textit{RTT}), goodput (\textit{BW}) and packet loss rates (\textit{loss}), \textit{site} has the strongest impact and, in fact, individually using one of the other three features for predictions lead to similar results as randomly selecting the configuration (random’s

\footnote{In Chapter 5.6.8, we discuss the role that the different types of features, such as website, network properties, etc., play in predicting the near-optimal configurations.}
performance shown in Figure 5.5).

While relaxing the control requirements is expected to reduce the proficiency of proposed mechanisms, our results, especially for dynamic configuration tuning, show that there is still significant benefits that could be achieved if there is control over a subset of relatively important layers. Similarly, if cross-tier control is not available, mechanisms like socket takeover that do not require such control can still be applied to reduce disruptions at scale due to the lack of inter-dependencies between the three Zero Downtime Release mechanisms. Effectively, there is a potential to partially apply the mechanisms to situations where the control requirements may not be fully met and still reap QoE benefits, although at a smaller magnitude than what is possible when the complete control is available.

7.3 Coordination across different organizations to improve applicability without a central control?

In the past decade, we have seen efforts where different organizations have joined hands to improve the content delivery for the end-users. A prime example of such joint efforts is CDN federations [295] that require coordination between multiple CDNs. In fact, Internet Engineering Task Force (IETF) working groups have systematically approached this effort to lay the ground work for coordination (CDNI) [386] and have published RFC7336 [406] that presents a framework to “specify interfaces and mechanisms to address issues such as request routing, distribution metadata exchange, and logging information exchange across CDNs”.

We believe that similar coordination efforts and standardization avenues like IETF can play a significant role towards reducing the control requirements for the proposed mechanisms, especially for the Zero Downtime Release mechanisms that require cross-tier control. If the protocol abstractions for downstream connection reuse and partial post replay can be standardized, it can provide a way for the infrastructures from the different organization to systematically coordinate with each other to conduct disruption-free restarts. However, any such efforts are expected to spawn a plethora of new challenges to solve, both from an administrative and research perspective, such as (i) trust assurance between two independent parties, (ii) resource cost economics and (iii) compatibility between different service implementations.
Chapter 8

Related Works

8.1 Innovations in the CDN Infrastructure

In the past few decades, a number of innovations have been made in the end-to-end CDN infrastructure. While it is infeasible to discuss every single paper due to their sheer number, below, we discuss some of the key papers concerned with the different operations in the end-to-end infrastructure.

8.1.1 Traffic Engineering and Load-balancers

Traffic engineering (TE) is a key component in the CDN infrastructure and is essential for maintaining performance, stability, availability and scalability of the infrastructure [90, 101, 91]. TE generally comprises two key mechanisms: (i) “end-user steering” systems are used for assigning end-user traffic to CDN-edge, and (ii) transport (L4) and application (L7) “load-balancers” (LB) are used for managing traffic within the CDN infrastructure.

**End-user Steering:** For steering requests to the edge, CDNs either use IP anycast [346, 58, 270, 334, 90, 414], or DNS-based steering systems [101, 354, 545]: the former relies on BGP for finding the closest edge to the end-users, while the latter cluster the end-users based on their local DNS or prefixes, and a global traffic management system keeps a live view of the Internet paths to generate optimal end-user mappings based on latency and operational constraints (e.g., bandwidth and CPU capacity). A number of studies have discussed limitations with IP anycast, such as BGP is oblivious to path latency [101, 237, 90] and load at an edge [307, 334], other have argued that anycast latency inflation impact is milder for CDNs than for services like DNS [281], and have further proposed tuning optimizations [334, 307, 270].

Since end-user steering systems inherently cluster similar user together (e.g., based on BGP-path distance, local DNS or prefixes), one can assume that connections at one a certain edge cluster may be similar and curating the serving stack at cluster-granularity may result in optimal performance. However, end-user steering systems do not guarantee homogeneity of user connections at the edge, e.g., users with different devices or last-mile connectivity (e.g., 2G and cable) may have the same nearest edge. Our work operates on the layers above steering and tackles heterogeneity among users mapped to the same edge. Further, our work
with minimizing downtime preserves cluster capacity when proxy servers are restarting and is complementary to the TE systems that balance edge cluster capacity by tuning request load. We also share the motivation for building systems that account for user-base heterogeneity in a fine-grained manner (e.g., prefix or DNS level clustering) and use of cross-layer metrics for generating the optimal decisions (e.g., cluster-level load, transport latency and throughput).

**Load-balancers:** The closest related work focuses on improving the design of software L4 and L7 load-balancers (LBs). Although our design is motivated by the recent trend of implementing load-balancing in software due to the availability, programmability and cost limitations of hardware LBs [161], our work stands in stark contrast to the related works. First, while our work borrows some ideas from previous works (i.e., consistent hashing [161] and socket migration [332, 54, 263]), our key novelty lies in designing abstractions that a commodity L7LB can use to minimize restart-related disruptions. A number of works have focused on improving throughput and packet-processing rate by leveraging kernel bypass [161], recent innovations in kernel such as XDP and eBPF [9], by designing new hashing algorithms [161], state-less LBs [41, 393], off-data-path LB [282], offloading consistency properties to data-path [59] and resource overhead optimizations [460]. Similarly, a number of articles and blog posts have discussed the design of LBs [468, 3, 382, 4, 510, 264, 532]. Our work focuses on a significantly different topic of tackling connection consistency and state issues that arise when an in-service L7LB restarts for updates or maintenance. Recently, a few proxies have been armed with disruption avoidance tools to mitigate connection terminations during restarts [332, 278]. HAProxy proposed **Seamless Reloads** [483, 263] in 2017 and socket FD transfer mechanism was added in 2nd half 2018 [5]. Envoy recently added **Hot Restart** [54] that uses a similar motivation. While our work borrows some ideas from previous works (consistent hashing [161] and socket migration [332, 54, 263]), our key novelty lies in designing abstractions that a commodity L7LB can use to minimize restart-related disruptions for a wide range of services (web, publish/subscribe and uploads), protocols (TCP, UDP, QUIC and HTTP/2), and connections (short-lived and persistent). Additionally, our work provides a first-time, hands-on view of the deployment of these techniques on a global scale.

A number of works have focused on improving performance from a TE-perspective by detecting congestion and latency issues and improving path selection from the edge [545, 182, 530, 444, 293]. Other works have focused on a DC-context and proposed novel designs for improving network performance, including switch software design [105], offloading host networking from individual server stacks to specialized hardware [183], dynamic adaptation for racks in DC by reconfiguring the network topology [100], optimizations for bulk transfers and latency-sensitive flows in DC by reconfiguring ToR switches [343], reconfigurability of high-bandwidth circuits in DC [360], and hardware-offload for the scatter-gather network workloads [31]. Our work with reconfiguring transport and application protocol configurations stands in stark contrast to these works: (i) we tackle a WAN setting and assume no control over path selection or network topology, (ii) instead of exercising control over both end-points of connections (as with DC setting), we are edge-centered, (iii) instead of introducing new data-path designs, our design complements the existing designs and improve their performance by dynamically adjusting TCP and HTTP configurations.
8.1.2 Network and Performance Monitoring

Monitoring the network conditions (e.g., link bandwidths, path latency) and performance (e.g., end-user QoE, error rates) is critical for ensuring robust TE and is, consequently, implicitly discussed in TE-related papers [545, 182, 101, 444, 293, 289, 289]. Further, a number of papers have proposed systems that explicitly deal with understanding and improving visibility from the CDN-end. NEL [84] at Google and Odin [91] at Microsoft leverage their client-side applications (e.g., Chrome, Edge, Office etc) for active and passive measurements from real-user devices to diagnose performance and availability issues. Similarly, others have proposed alternate designs to gain performance and network visibility, e.g., characterizing web performance and user-achievable goodput from the server-side at Facebook (Schlinker et al. [443]), in-band telemetry for end-to-end network visibility (LightGuardian [555]) and a novel metric for capturing user-perceived availability for Google’s G-Suite services (Hauer et al. [218]). Additionally, other studies have explored longitudinal measurement of GCP connectivity (Mok et al. [353]), user QoE measurements for public edge deployments (Xu et al. [536]), analysis of financial trading networks (Bhattacherjee [70]), usage of Tier-1 ISPs and other large transit for cloud providers (Arnold [46]), study of disruptions and outage detection from edge (Richter [425]) and fault detection and diagnosis in DC software CDN plane (Roy et al. [428]).

Our works with configuration tuning and downtime prevention leverage the existing systems for monitoring network characteristics (e.g., Apache’s TE logs and TCP plugin [450, 116]) and request performance (e.g., metric discussed in [444]). Since device-specific resources (e.g., memory usage) cannot be readily monitored from the CDN-end, we develop a client-side measurement technique for monitoring the memory dynamics of page loads.

8.1.3 Caching Systems

A number of systems have explored improving cache performance from systems and cache admission/eviction algorithmic perspective. CacheLib [64] presents the design of a general-purpose caching engine at Facebook that reduces the overall effort required to deploy, maintain, and scale caches. AdaptSize [67] proposes a tuning scheme for size-aware caching in CDNs using a Markov model to incorporate request rate distributions and object sizes. Song et al. [469] uses ML to approximate offline Belady MIN (often used as Oracle for measuring caching performance). Shen et al. [459] leverages information bound references (IBRs) [35] to collapse multiple related video cache entries into a single one to optimize hit-rates. Flores et al. [185, 186] presents the design of caching simulator used at EdgeCast and presents an empirical comparison of different caching algorithms. Sundarrajana et al. [478, 479] presents a theoretical foundation for modeling CDN caches and leverages the model to improve hit-rates and midgress traffic between CDN PoPs. Similarly, a number of systems focus on optimizing latency of caches [66, 49].

Although our work does not directly address caching, we leverage the motivation used by data-driven cache tuning systems for optimizing the performance and reliability of heterogeneous connections. In caching system’s context, such heterogeneity is often leveraged on workload-granularity to design the caching algorithms that match the diverse request patterns of different workloads, e.g., video, OS updates, web, etc.
8.2 Continuous Code and Configuration Release

Since CDNs aim to release new features, bug fixes and security updates as soon as possible, a number of studies have discussed continuous release and its role in today’s development cycles.

8.2.1 Continuous Release Adoption

Humble et al. [240] presents the design goals for continuous release, associated challenges and case-studies to highlight its impact on productivity, cost and code quality. Adams et al. [15] discusses code release pipelines adopted by major companies and proposes a checklist of points-of-concern for future researchers. Savor et al. [440] presents a measurement study of code releases at OANDA and Facebook and shows its impact on productivity and code quality. Further, reports from SE Daily [122, 121], Dyck et al. [157] and Adams et al. [14] compile interviews with industry experts and highlight their perspectives on the benefits and limitations of continuous release, cultural change, required skills, and future research problems.

Our work with continuous release advances the related works by presenting more-recent measurements about the practice and the impact of releasing code and configuration updates at a global scale. We further provide an analysis of impact of continuous release on diverse services (e.g., TCP, QUIC, MQTT, HTTP uploads), and propose system and protocol abstractions that help in minimizing downtime and disruptions at scale.

8.2.2 Software and hardware updates

CDN operators traditionally rely on over-provisioning the deployments and incrementally release updates to subsets of machines in batches. Examples of such techniques include blue/green deployment [451] and rolling updates [288]: the former relies on over-provisioning by maintaining two identical environments where one runs last-stable version while the runs newer code, while the latter incrementally updates the machines in batches. These techniques are supported by frameworks like AWS CodeDeploy, Kubernetes, NGINX, Envoy etc. Orthogonally, there are efforts to redesign the load-balancing plane to eliminate the over-provision and disruption-related limitations of these approaches by running multiple instances of an application on same hardware and seamlessly migrating traffic from one application to another (typically called hot restart [53]). This feature is supported in proxy applications like HAProxy [4, 483, 323, 263], Envoy [3, 54], Proxygen [468] etc. There are also efforts to design languages with built-in support for headless updates such as Erlang [40]. However, this approach is not supported by most common languages and only a trivially small number of our services are written in such a languages.

The challenge of updating hardware or software safely in production networks has also been tackled in research literature. AutoPilot [246] and Gandalf [306] discusses Microsoft’s approach for updating hardware or software in DC, and monitoring fault signals to catch bad roll-outs before widespread outages. In data-center context, a number of studies have discussed updating switch software and hardware without any associated disruptions. Reitblatt et al. [422] tackles consistent switch configuration updates that preserve network robustness and avoids black-holes. zUpdate [315] proposes a scheme for managing network migrations without any associated downtime or packet loss. Chi-Yao et al. and Xin et al. [231, 257] discusses network updates
in WAN context and proposes schemes for compartmentalized updates to the infrastructure. Janus [28] highlights the performance and downtime risks of updating network hardware and software in data-centers, and proposes an domain-specific algorithm for planning multi-switch updates without disruptions.

The key difference between the traditional approaches and our work is that we do not require any over-provisioning, and our optimizations are able to tackle downtime and disruptions for a wide range of services (i.e., TCP, QUIC, MQTT, HTTP uploads), that traditional approaches do not tackle. In essence, our work addresses the limitations with the traditional approaches. The research works in literature, while related, tackle the problem from a different angle: either they focus on catching bad roll-outs quickly and rolling back efficiently to last stable version, or focus primarily in a network switch context and proposes techniques for safe traffic migrations. Our work focuses on the L5 and L7 components (load-balancers and application servers) and propose domain-specific optimizations for improving roll-out efficiency.

8.2.3 Tools for Simplifying Configuration Updates

To aid operators and automate the configuration tuning workflows, a number of tools have been proposed. Their key purpose is to reduce human efforts and automate the diagnosis of errors and configuration update vulnerabilities. Robotron [480] and Tang et al. [482] discuss deploying configurations at Facebook and present a suite of tools for deploying, monitoring and translating high-level design intent into low-level configurations. PCHECK [537] proposes a system for early detection of configuration errors by analyzing the source code and automatically generating configuration checking code. Further, Config2Spec [74] and Spex [538] proposes tools for automatically synthesizing a formal specification of a network given its configuration and a failure model, and inferring configuration requirements from source code, respectively.

These works focus on automatic testing of new configuration updates and their deployment. Our work does not propose any new configurations update systems, and leverage existing work [480] for deploying the updates. Instead, the focus of our work is minimizing disruptions and downtime after the configuration updates have been deployed.

8.3 Configuration and Network Tuning Systems

Systems e.g., data-bases, video, cloud VMs etc., and network protocols, e.g., TCP, HTTP etc., provide users and operators with a wide range of configuration options to choose from. As performance of these systems and protocols depend on careful tuning of the configurations according to the needs of workloads or networks, a number of papers have focused on their configuration tuning. Below, we discuss some of the key works.

8.3.1 Algorithms for Configuration Tuning

A number of different techniques are employed for configuration tuning, i.e., finding the configuration that maximizes a target utility function (e.g., performance, cost, latency etc.) under a given scenario (e.g., workload, network etc.).
Some CDNs opt for an experiment-based approach, where a range of configurations are tested in a test-bed, simulation environment or for production traffic, and good configurations are identified by analyzing the traces, e.g., TCP initial congestion window tests [152, 27, 432, 187], congestion control (CC) tests [433, 523, 93, 92, 95, 96, 445, 229], HTTP tests [27, 557, 558]. Further, specialized infrastructure tools and test-beds have been proposed to improve the efficiency of such experimentation, e.g., Kraken at Facebook [511], Odin at Microsoft [91], Mahimahi and EyeOrg for web [378, 506], Pantheon for CC [543], Puffer for video [541]. Such experimentation and their implications are extensively discussed in articles from CDNs like Facebook [199], Akamai [23, 22], EdgeCast [456, 37] and Cloudflare [485, 341, 213, 412]. Based on the measurements, operators either use the top-performing configurations homogeneously for all user-base, or certain coarse-grained classes of connections (e.g., cellular traffic) or workloads (e.g., video traffic).

A key limitation of experiment-based approach is that the results may not be generalizable over time, as the underlying conditions (e.g., network, workloads, devices etc.) evolve. Further, there is a human-effort overhead associated with repeating the experiments and re-drawing conclusions. To tackle these limitations, some CDNs opt for dynamic, data-driven approaches that broadly fall into two classes.

The first class involves algorithms for modeling the performance-configuration curves offline, through traces generated in simulators, test-beds or in-production. A plethora of modeling algorithms are available, ranging from statistical techniques (e.g., clustering [248, 512, 312, 262, 268, 216, 504], Gaussian distributions [210, 29, 324, 287], Markov models [25, 476]) to machine learning (e.g., decision trees [417, 447, 520], random forests [448, 437, 540], neural networks [529, 546]). The offline-generated models are then applied in real-time to infer optimal configurations based on a set of features (e.g., user network, application workload). The models are refreshed with new traces over time to ensure their representativeness.

The second class takes an online approach towards finding the optimal configurations. A number of such algorithms have been proposed, ranging from guided exploration (e.g., Bayesian optimization [233, 404, 198, 79, 300, 16, 29, 310, 435], A/B testing [383, 471], hill-climbing [71, 533], genetic algorithm [311, 351]), multi-armed bandit [513, 322], exploration/exploitation [553, 255, 256, 287, 234, 303] to reinforcement learning [305, 251, 431, 384]. These systems typically take a cross session/workload view and model the performance-configuration curves from the real-time (and historical) data to infer optimal configurations.

The two classes of algorithms have their own pitfalls, and a number of papers have discussed a comparative analysis of such learning techniques in systems context [71, 127, 72, 73, 61]. While offline techniques are generally easier to deploy, they are limited in their ability to tackle Internet dynamics [305]. On the other hand, online techniques often face biasness issues [61], e.g., Spang et al. [471] discusses the biasness implications of A/B testing at scale and show that biasness can have multi-dimensional impacts on results, ranging from misdirection to requiring tons of samples to generate meaningful results. To tackle these limitations, a number of systems take a hybrid approach (i.e., complement offline models with online data) and modify the algorithms to tackle their own domain-specific constraints.
8.3.2 Data-base, Cloud-computing and Video Tuning

Due to sensitivity of performance to configurations in systems, configuration tuning is performed for a variety of data-bases and cloud-computing. Consequently, a number of systems have been proposed to tune the configurations of Map-reduce [228, 304], Databases [151, 504, 403], and cloud computing [29, 227, 310, 556, 71, 350, 516, 156, 457, 439, 72, 259, 233, 540, 512, 235, 234] to improve application-specific performance. The key motivation behind these works is to leverage heterogeneity in configurations to curate specialized rules the suit the needs of provided workloads or operating conditions (e.g., cloud hardware, database workload etc). Some of these approaches leverage modeling for static workloads where performance metrics do not change with time [29, 151] and exploit the optimal configuration choice for the provided workloads. On the other hand, others take an online approach and leverage online algorithms to learn optimal configurations in an online manner and adapt over time or for dynamic workloads [504, 310].

A number of works have focused solely on optimizing the client-side dynamics of video streaming. Client-side video applications use ABR (Adaptive BitRate) algorithms to dynamically decide a bitrate for each video chunk (e.g., 4 second video chunk) based on client-side features, e.g., network throughput, buffer capacity etc [24]. ABR-tuning systems typically work in two phases. The first phase involves offline modeling (through a simulator or past traces) to map client-side features to optimal ABR decisions, e.g., DeepRL-based CNN training in Pensieve [329], exhaustive exploration of space in Oboe [25], clustering past sessions in CS2P [476], crowd-sources QoE-sensitivity evaluation in Sensei [552]. In the second, phase the model is applied to clients at real-time to optimize ABR performance by identifying client-side features (e.g, estimation through HMM (Hidden Markov Model) [476], network change detection algorithms [25]) and using them as input features to the model (e.g., neural network [329]). Alternatively, systems have been proposed to improve CDN selection and requested bitrates. Systems like CFA [255] and Pytheas [256] exploits similarity in client sessions at the CDN edge and uses either modeling or online exploration and exploitation for groups of similar sessions to optimize bitrate and CDN selection decisions. C3 [197] proposes a split-plane architecture for optimizing bitrate and CDN selection that is scalable and applies to diverse clients, while VDN [361] proposes a centralized control-place for optimizing the delivery of live video. In spite of all these works, practically improving video performance is still a challenge. Yan et al. [541] conducted randomized controlled trials of popular video-streaming algorithms and observed the challenges with emulating diverse Internet paths in offline training and heavy-tail user and network behaviors led to them performing similar to simple buffer-based techniques.

Orthogonally, other systems have optimized other avenues in the design space, ranging from client-side quality enhancement of low-quality video through DNN (Yeo et al. [546]), transport optimizations (Minerva [368], Vargas et al. [505]), coupling transport with video codec (Salsify [188]), server-side caching (iProxy [459]), client-side connection and rendering issues (Ghasemi et al. [201]) to fine-grained parallelized encoding through cloud functions (ExCamera [189]). Some systems have focused on tuning configurations for analytics for live video, e.g., VideoStorm [549] trades-off quality through tuning video knobs (e.g., resolution, frame-rate etc.) to assign higher resources to time-sensitive queries, Llama [426] proposes a cost-based optimizer to assign configurations across heterogeneous hardware for video analytics.
8.3.3 Network optimizations and tuning systems

Network performance optimization is a rich field and a number of studies have focused on optimizing delivery of content, latency/bandwidth improvements, and tuning for transport and application protocols.

Optimizing delivery by improving bandwidth usage and reducing latency for network flows is discussed in a number of TE systems [545, 182, 101, 444, 293, 289, 313]. These systems integrate the configuring decisions of individual or multiple elements of infrastructure (edge, WAN, DC etc.), leverages control over infrastructure, domain-specific modeling and multi-session view of flows to optimize delivery.

At the transport and application layers, a number of optimizations have been proposed ranging from novel congestion control algorithms (CCA) (BBR [95, 96, 445, 229], COPA [47], PCN [104]), ML-powered and learning-based congestion control (TCP ex machina [529], PCC [149], PCC-Vivace [150]), transport and application protocol configuration tuning (dynamic CCA switching [102, 202], initial congestion window tuning [384, 385, 383, 187], RL-based CCA [305, 251]), dynamic HTTP/2 PUSH [557, 558, 267]), dynamic compression strategies (Flexiweb [464]), to modeling of network flows (Econ [93], Ware et al. [523], Schapira et al. [442], LRP [481], Cao et al. [92], Vargas et al. [505]).

8.4 Web Performance Optimizations

Web performance optimizations are aimed towards reducing the page load time (PLT) or other performance metrics like SpeedIndex [340], first-contentful-paint [338], time-to-interactivity [171] etc. Below, we discuss a few different direction to improve web performance.

8.4.1 Optimizing delivery

Optimizing the delivery of web content (e.g., HTML, CSS, JavaScript, media objects etc.) is critical, as network is a major bottleneck for web performance. Consequently, a number of optimizations have been proposed including: (i) server/proxy-based accelerators (OperaMini [349], Oblique [280], Vroom [430], WatchTower [380]), (ii) reducing network overhead (e.g., compression [173, 167, 308], image minification [276, 106, 167, 308, 397], browser caching [347, 377, 205], debloating use-less content [535, 509, 225, 221, 528, 168, 517, 32]), (iii) protocol enhancements (e.g, QUIC [212], HTTP/2 push [557, 267], HTTP tuning [520, 165, 27], TCP tuning [152, 383, 19]).

Other works have explored tracking dependencies in website structure to optimize the delivery of objects that directly impact a target performance metric (i.e., objects on critical path on page load process). These works cover inspection of webpage complexity and dependencies [518, 371, 85, 375] and dynamic fetch prioritization and scheduling [88, 331, 309, 375, 522]. Some other works have taken a different approach towards deciding the delivery prioritization of objects, e.g., prioritizing objects that contribute to above-the-fold content [171], top view-ports [342, 274] or user-centric utility [275, 401, 437].
8.4.2 Resource overheads and JavaScript optimizations

Hardware resources (e.g., CPU, memory, battery) play a critical role for mobile web performance [373, 296, 94, 123, 20]. A number of studies have explored the role of device computational resources on mobile web performance and observed computation to be a significant performance bottleneck [518, 371, 123].

JavaScript is observed to be a key bottleneck to web performance due to its high CPU and memory overhead [179, 398, 373, 250, 400]. Consequently, a number of optimizations have been proposed ranging from understanding of JavaScript APIs [170, 438], removing non-critical JavaScript [168, 292, 347, 509, 147] to the use of cloud-based browsers to offload JavaScript computations from the client device [349, 466, 454, 253]. Some other studies have explored partially offloading JavaScript computations to server or 3rd-part proxies to assist low-end devices [376, 522, 280, 430, 380]. Additionally a number of rendering optimizations have been proposed to simplify the page load process and reduce the overhead of client-side JavaScript. These include static rendering [250], server-side rendering [400], pre-rendering and hydration [399, 398], and general rendering optimizations [146, 224, 215, 302, 138, 301]. These optimizations either fully or partially render the website DOM at the server-side, thus saving the client browser from computational work and network fetches. Orthogonally, a number of studies have focused on the battery and energy optimizations aspects of mobile web [507, 508, 94, 551, 97, 103].

Our work differs from these related works in several key ways. While none of these works focus specifically on JS memory overhead, the primary focus of our work is the design and implementation of a novel tool (JS-Capsule) to address the limitations with existing measurement methodologies. Unlike our prior work that focus on device memory charateristics around the world and identifies JS as a key source of memory overhead, this paper tackles the challenge of measuring and attributing memory on the fine-granularity of JS functions, a key challenge that was not addessed in the prior work due to the coarse-granularity of the measurement approach. In contrast to [123], that focuses on QoE dynamics for different amounts of available memory and does not quantify the memory overhead of today’s web, we present a characterization of JS memory for Alexa top 1K websites and identify various sources of memory overheads. While [94] focuses on browser events in the context of energy usage, our measurements are for a completely different domain and our measurement technique goes a step further by attributing browser events to JS functions. In contrast to [416], instead on focusing on the web-memory performance aspects, we dissect why the JS memory overhead exists, with a much more fine-granular approach.

8.4.3 Browser Optimizations

Recent works have proposed several browser optimizations to improve performance and resource usage. [495] discusses lazy parsing schemes for V8 to improve computational and memory usage. [75] discusses heuristic-based interventions in Chrome to disable advertisement JS. Similarly, others have focused on ad and tracker blocking schemes for browsers [172]. Similarly, Chrome developers have proposed numerous optimizations, including “trimming the fat” by removing excess JS functionality [179], PRPL pattern with guidelines for pre-load, rendering, pre-caching and lazy load [146], advising the best practices to improve the layout of webpage rendering [68], introducing a custom regulatory layer in front of DOM to improve DOM
batching [177, 178] and CLS [111] for measuring layout shifts and visual stability. Further, researchers and V8 developers have proposed number of profilers to understand V8 efficiency [11, 495, 170, 441, 539].

The context of our work stands in stark contrast to these works. First, instead of just focusing on V8, ads-/trackers or specific browser interventions, our study analyzes the cross-component aspects of JS and discusses the implications of direct (i.e., V8) and indirect memory overhead. Instead of focusing on a particular type of JS such as advertisements, we analyze all sources of JS in a website. While the Chrome recommendations generally tackle performance improvements, our work quantifies the different types of JS memory allocations and qualitatively assess their root-cause in order to inform the selection of appropriate recommendation/optimization and highlight the deficiencies of current widely-used optimizations in holistically saving memory. Further, our layout score is different to CLS as (i) CLS is a visual score and only considers positional changes for the existing visual parts of the webpage in the current view-port, whereas a re-layout may not change the position but only the style or the change might be outside the current view-port, and (ii) CLS only computes a single value for the entire webpage, whereas we required a metric that could be calculated for each capsule.
Chapter 9

Conclusion and Future Directions

In this chapter, we conclude the dissertation by summarizing the key contributions of the works and outlining the future research directions to help the community in further combating the challenges that arise due to heterogeneity in user-base and services.

9.1 Summary of Contributions

In this dissertation, we make the case that the rising heterogeneity in user-base and services has spawned a new set of challenges toward maximizing performance and availability. Traditionally, CDNs opt for generalizable solutions to maintain simplicity and reduce the cost of infrastructure. However, such solutions are unable to address the domain-specific requirements of the diverse user-base and the services and, consequently, lead to sub-optimal QoE. While CDNs can design and deploy specialized solutions for each class of heterogeneity, unfortunately, in the realm of traditional designs, such specialized designs require a significant cost, either in terms of developer effort or resource overheads, thereby presenting a dilemma and friction between either maximizing QoE or maintaining the least-cost of the infrastructure.

This dissertation explore the CDN design space to improve the QoE vs least-cost trade-off that arises due to heterogeneity outside of a CDN’s purview. While a CDN may not have direct control over the source of heterogeneity, the eventual impact of heterogeneity manifests itself in the form of different types of state across the layers, e.g., transport network properties of a connection, application-level state for diverse services, client-side resource allocations for the page load. Using the state manifestation of the heterogeneity, the dissertation proposes “CDPlane”: a collection of data-path and control-path design optimizations, and makes the following three contributions.

Configanator addresses web performance challenges associated with the end-user heterogeneity by designing a framework for tuning data-path cross-layer (transport and application layer) configurations on a per-connection basis. To control the configuration flexibility, it proposes the design of control-plane and a novel algorithm to tune server-side cross-layer network configurations in a principled, online and low-overhead manner.
ZeroDowntimeRelease explores data-path system abstractions that allow fast-paced code updates to Hyperscaler infrastructure without any associated downtime. We propose cross-tier and cross-layer optimizations to introduce flexibility in the serving infrastructure layer, i.e., application servers and HTTP proxies, and further propose changes to application protocols to support code update flexibility without any disruptions.

JSCapsules introduces a measurement technique to gain cross-layer visibility into the JavaScript’s impact on client-side resources. Leveraging this techniques, we conduct a measurement for Alexa top 1K websites, characterize their JS memory overhead and identify avenues for improvement.

While the traditional designs operate within the context of individual layers due to the modularity and isolation principles, CDPlane trades off generality in favor of finding a better cost-QoE trade-off. While the domain-specific design of the algorithm to combat network dynamics eliminates the cost of manually curating networking stack configurations, the proposed system and protocol abstractions for disruption-free updates only extend the traditional designs and does not require a specialized redesign of services from the ground up. Further, the analytics technique directly identifies the source of memory bloat and uncovers potential avenues for optimizations, reducing the developer effort required to inspect the large space of JavaScript functions, libraries, and their interactions with the browser components. Taken together, these building blocks aid a CDN in extending their design to improve QoE for the diverse services and user-base, and optimize the tussle in a low-cost manner.

9.2 Future Work

The motivation behind the design of CDPlane can be extended to a wide range of web stack layers to improve QoE for the heterogeneous users. Below, we discuss two such directions.

9.2.1 Broader configuration flexibility

Next-gen congestion controls (CC) like BBR and cross-layer protocols like QUIC provide a higher level of flexibility than what is available for traditional TCP, e.g., legacy congestion controls like Cubic do not maintain an extensive model of the connection’s underlying network and instead rely on signals (e.g., packet loss) to infer connection events.

However, the picture is quite different for next-gen protocols. Specifically BBR, unlike the legacy CCs, maintains a comprehensive internal model of the connection, constantly probe to infer the network conditions, and provide a slew of configurations (mostly baked into the kernel) to an operator to modify the protocol behavior. The presence of the inherent network model provides an opportunity where deeper visibility can be gained into an end-user’s network state and directly leverage the state changes as a feedback signal for auto-tuning BBR’s internal configurations. While on one hand such flexibility opens avenues for extending the scope of Configanator’s configuration space into deeper CC mechanics, it further provides an opportunity where the control-plane logic can be partially baked into the data-plane itself and leverage the internal model/probing to auto-tune configurations directly from the data-plane.

Similarly, QUIC’s control over the application and transport layers can be leveraged to add interfaces between the two layers to optimize delivery, extend HTTP functionality (e.g., prioritization) to transport and
add ability to coordinate the two layers and to resolve any transport level bottlenecks (e.g., write buffers), and vice-versa. For instance, in case of packet loss, TCP follows a rigid behavior where the sending the lost packets are prioritized before sending new data. However, QUIC’s packet loss behavior is much more customizable and the implementation can be extended to directly incorporate signals from the application to tune the loss recovery behavior, e.g., instead of re-transmitting data, prioritize packets for a higher priority object. Since CC limits the transport congestion window at loss event, tuning the loss recovery behavior during the small congestion window phase can potentially have significant benefits, based on the eventual QoE impact of the object’s loading priority for the webpage.

9.2.2 User-centric layers and utility scores

A webpage is a collection of individual web objects and the different types of web objects (e.g., HTML, CSS, JavaScript, images etc.) serve a distinct purpose, e.g., HTML defines the fundamental structure of the webpage, CSS defines the styling rules, JavaScript is used to add dynamic content and interactive functionality. Consequently, irrespective of the type, each web object is aimed towards providing a certain utility to the webpage. From a user’s perspective, this utility is traditionally categorized into two classes: visual (objects that impact the visual appearance of webpage) and interactive (objects that add a functional interactivity to the webpage). The extent to which the the two classes are used in webpage highly depends on the type and business requirements of the webpage, e.g., while wikipedia.org articles pages tend to be plain in appearance and relatively static, the best-sellers page in amazon.com is much more rich in content, appearance and functionality.

While a webpage is constructed on the granularity of web objects, users do not always necessarily view or interact with individual web objects. Instead, they view and interact with a processed output: the browser processes the multiple individual objects, creates an internal representation of the webpage (e.g., DOM,
CSSOM, layout tree etc.), renders the internal representation on screen, and attaches event listeners defined in HTML or JavaScript. To illustrate, Figure 9.1 shows the multiple view-ports for espn.com. In the 1st view-port, the top-story components’ structure is defined in HTML template (section tags) and JavaScript functions make network requests to fill-in the template text/image objects. The browser also leverages the CSS objects for applying the styles to the top story components. The top-story component — a culmination of processing multiple individual objects — represents an individual information block that a user may be interested in viewing or interacting with. Thus, from a developer’s perspective, while the webpage consists of a number of objects, the users see the webpage as a collection of blocks, and each block provides a certain visual (i.e., painted objects within the block) and functional (i.e., event listeners attached within the block) utility to the user.

Due to the discrepancy between the developer’s and the user’s perspectives, there is a disconnect between how the webpages are designed/optimized and how they are viewed. From a user’s point-of-view, a well-performing webpage should load the blocks fast, i.e., the blocks load state should be ready in the minimal amount of time so that users can view and interact with the blocks. In fact, studies show that users interests may vary across different parts of the webpage [275, 437], thus a webpage that loads the “interesting” parts of a webpage (or blocks) faster would be preferable to the users. However, developers often take a “performance-metric centered” approach towards optimizing the performance of their webpages: they identify the subset of objects that directly impact the target performance metric and optimize their delivery and processing. It is not quite apparent whether this approach directly translates into improving the user-centric performance.

Assuming we can develop an abstraction that can connect the user’s and developer’s perspective and provide a quantitative utility score to the different blocks in a webpage, we can directly evaluate the user-centric QoE outcomes of different types of network, computational and resource optimizations for the webpages. In fact, such an abstraction can provide a new flavor to some of the key web optimizations proposed in the recent few years such as (i) how to prioritize delivery that contributes to the high utility blocks?, (ii) how to optimize the browser layers to directly consume the utility information and improve the scheduling for the loading the high utility blocks quicker? and (iii) can we trade-off low utility blocks in order to save computational and memory resources for the page load?
Bibliography


[34] Amp.dev. Amp supported browsers. https://amp.dev/support/faq/supported-browsers/.


[130] DEVELOPERS, A. Memory allocation among processes.


[144] DEV@TRAFFICSERVER.APACHE.ORG. Ja3 fingerprint plugin. https://bit.ly/3iTg70U.


[146] DJIRDEH, H. Apply instant loading with the prpl pattern.

[147] DJIRDEH, H. Reduce javascript payloads with code splitting.


[190] FOUNDATION, L. Open vSwitch.


[217] **HAT, R.** Interpreting /proc/meminfo and free output for red hat enterprise linux.


[224] HEMPENIUS, K. Use lazysizes to lazy-load images.


[238] HULCE, P. Long task attribution in lighthouse. https://docs.google.com/presentation/d/1ss1nZi2MyyK1Nb6LoS_vTlcvshkUR0IUekacq4mZMhs.


[265] JUN, B., BUSTAMANTE, F. E., WHANG, S. Y., AND BISCHOF, Z. S. Amp up your mobile web experience: Characterizing the impact of google’s accelerated mobile project.


[271] KAYSER, B. What is the expected distribution of website response times?

[272] KEARNEY, M. Memory terminology.

[273] KEARNEY, M. Record heap snapshots.


[335] MDN. Cross-origin resource sharing (cors).


[339] MDN. Object prototypes.


[345] A first look at existing metrics w/ and w/o memory-infra. https://docs.google.com/presentation/d/1OyxyT1sfg501A36A7ibZ7-bBRXI1kVlVwCOW9qAmM_0.


[349] MINI, O. Opera mobile browsers. Opera Software.


MQTT.ORG. Mq telemetry transport, machine-to-machine (m2m) connectivity protocol. http://mqtt.org/.


NEWMAN, S. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2015.


NOVAK, A. Going to market faster: Most companies are deploying code weekly, daily, or hourly. https://newrelic.com/blog/best-practices/data-culture-survey-results-faster-deployment.


[400] OSMANI, A. A netflix web performance case study.


[489] Chrome javascript timer throttling: Google’s tests show it saves up to 2 hours’ battery life. https://www.zdnet.com/article/chrome-javascript-timer-throttling-googles-tests-show-it-saves-up-to-2-hours-battery-life/.


[496] Trace event format. https://docs.google.com/document/d/1CvAC1vFfyA5R-PhYUm5Q0QtYMH4h6IOnSsKchNAySU/.


