Solutions for Link Rot on the Modern Web

by

Jingyuan Zhu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2025

Doctoral Committee:

Professor Harsha V. Madhyastha, Co-Chair
Associate Professor Ryan Huang, Co-Chair
Associate Professor Steve Oney
Professor Atul Prakash

Jingyuan Zhu
jingyz@umich.edu
ORCID iD: 0000-0002-3724-3298

© Jingyuan Zhu 2025

To my family and friends.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Harsha V. Madhyastha. His responsible and valuable guidance has greatly shaped my work and my growth as a researcher. I have always appreciated how he pushed me to think more deeply to refine our work, while also supporting me during challenging times with kindness and understanding. Working with him has been one of the most rewarding and memorable experiences of my graduate life.

I would also like to thank my dissertation Co-Chair Prof. Ryan Huang, and my committee members: Prof. Atul Prakash and Prof. Steve Oney, for generously dedicating their time to review my work and offer insightful feedback.

I'm deeply grateful to my collaborators–Ayush Goel, Huanchen Sun, Vaspol Ruamviboonsuk, Anish Nyayachavadi, and Jiangchen Zhu. Their creativity, diligence, and teamwork made every project enjoyable and productive. In particular, stimulating discussions with Ayush and his generous sharing of ideas has provided me with a lot of insights, while Huanchen's help in running critical experiments—often at the very last minute—saved me on multiple occasions.

I would also like to thank my friends—Yuhan Chen, Xiuneng Lu, Xisong Weng, Yu Zhang, and Peiyi Gu—for filling my life outside of research with laughter and warmth. The time spent together has been a constant source of encouragement whenever challenges arose. I would also like to thank my colleagues in BBB 4929—Yuan Yao, Muhammad Uluyol, Chris Baik, Joseph Lee, Abhinav Nakarmi, Zhenning Yang, Archit Bhatnagar, and many others—for the fun and chatting both inside and outside the office.

Most importantly, I would like to express my heartfelt thanks to my family. My parents, Yuhong Shang and Xuemin Zhu, have been the foundation of all my achievements. Their love, patience, and unwavering belief in me have given me strength at every step. I'm also thankful to my aunt and uncle—Yujuan Shang, Yuqing Shang, Aihui Zhu, Fuhui Zhu—and my grandparents, Yaojin Shang and Jianrong Zhu for their constant care and encouragement. Their support and affection have been my greatest source of comfort and motivation throughout this journey.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

LIST OF TABLES viii

ABSTRACT ix

CHAPTER

1 Introduction 1

1.1 Web archives: Today's solution for link rot and their limitations 1
1.2 Thesis and Contributions 4
1.3 Organization 7

2 Background and Related Work 8

2.1 Background 8
2.1.1 Prevalence of link rot 8
2.1.2 Web archival 10
2.1.3 Limitations of web archives today 14
2.2 Related work 17
2.2.1 Broken links 17
2.2.2 Web crawling and caching 18
2.2.3 Program testing and debugging 19

3 Reviving Dead Links on the Web with Fable 21

3.1 Motivation 21
3.2 Overview of Fable 24
3.3 Design 28
3.3.1 Backend design 29
3.3.2 Frontend design 34
3.4 Evaluation 38
3.4.1 Coverage and Accuracy 39
3.4.2 Efficiency 46
3.4.3 Utility of Aliases 49

3.5 Summary . . . 51
4 Toward Better Efficiency vs. Fidelity Tradeoffs in Web Archives . . . 52
4.1 Dataset and Approach . . . 53
4.1.1 Data collection . . . 53
4.1.2 Fidelity assessment . . . 54
4.1.3 Static vs. Dynamic . . . 57
4.2 Reusing resources across crawls . . . 58
4.2.1 Reusing resources from other pages . . . 58
4.2.2 Reusing resources from other times . . . 59
4.3 Accounting for inter-resource dependencies . . . 60
4.4 Handling resources which cannot be reused . . . 64
4.5 Efficiency–Fidelity Tradeoffs . . . 67
4.6 Summary . . . 67
5 Detecting and Diagnosing Errors in Serving Archived Web Pages . . . 68
5.1 Overview . . . 69
5.1.1 Goals . . . 70
5.1.2 Challenges . . . 70
5.1.3 Guiding observations and approach . . . 72
5.2 Design . . . 74
5.2.1 Identifying fidelity violations . . . 74
5.2.2 Pinpointing root cause of fidelity violations . . . 77
5.3 Implementation . . . 82
5.4 Evaluation . . . 83
5.4.1 Dataset . . . 84
5.4.2 Fidelity violation detection . . . 85
5.4.3 Pinpointing of erroneous rewrites . . . 90
5.4.4 Fixing commonly pinpointed errors . . . 91
5.4.5 Efficiency . . . 93
5.4.6 Generalizability . . . 96
5.5 Summary . . . 97
6 Conclusion and Future work . . . 98
6.1 Thesis Contributions . . . 99
6.2 Long-term applicability . . . 100
6.3 Future Work . . . 101
6.4 Summary . . . 104
BIBLIOGRAPHY . . . 105

LIST OF FIGURES

FIGURE

1.1 Example of the Wayback Machine's usage. 2

2.1 Links break a few years after they are posted, and they point to a wide variety of external sites. 9
2.2 Screenshots when loading copies of an example web page [46]. 12
2.3 Example of dynamic rewriting of resource URLs. 13
2.4 Screenshots when loading pages from two example pages. 16

3.1 Many URLs on a site go dead together. 23
3.2 Overview of FABLE's architecture. 26
3.3 FABLE's browser add-on demonstration. 27
3.4 Comparing historical redirections to identify which redirections are soft-404s. . 30
3.5 Contrast between precise and coarse-grained characterizations. 31
3.6 Programming by example in action. 36
3.7 FABLE significantly improves coverage and accuracy. 40
3.8 Efficiency of FABLE backend. 48
3.9 Median latency incurred at the FABLE frontend as a function of whether or not it finds the alias and the method it uses when successful. 49

4.1 Fraction of static crawls have no missing resources and do not violate fidelity. . 55
4.2 Demonstration and example of how bundling causes a JavaScript's URL to change. 61
4.3 Increase in the fraction of static crawls after bundling-aware patching. 62
4.4 An example page where all the tokens in the URL can be traced. 64
4.5 Increase in the fraction of static crawls with crawling augmentation. 66

5.1 A web archive's workflow augmented with FIDEX. 69
5.2 Examples of dynamic pages. 71
5.3 Illustration of browser's critical rendering path. 72
5.4 Example inspired by the code found on skype.com in Dec 2024. 75
5.5 How FIDEX confirms whether a runtime error led to violation of fidelity. 79
5.6 Example of a fidelity violation with silent errors. 80
5.7 Example of FIDEX detecting silent errors. 81
5.8 FIDEX versus three baselines. 87
5.9 Impact of fidelity issues detected by FIDEX. 89
5.10 Clustering of FIDEX's pinpointed errors. 91
5.11 Breakdown of the impact of my modifications to pywb. 93

5.12 Comparison of crawling throughput. 94
5.13 FIDEX's runtime. 95

LIST OF TABLES

TABLE

2.1 Sizeable fraction of external links are broken 9

3.1 Example of using archived copies in case the page has an alias. 22
3.2 Example of a group of 3 broken URLs, their titles and corresponding aliases. . . 25
3.3 Summary of the techniques used in FABLE. 28
3.4 Step 1 of matching broken URLs in a directory to their aliases. 32
3.5 Step 2 of matching broken URLs in a directory to their aliases. 33
3.6 Example demonstrating challenges in using PBE. 34
3.7 FABLE's success rate in finding the alias as a function of how the URL is broken. 45
3.8 FABLE's success rate in finding the alias for a URL as a function of how long it has been since that URL stopped working. 45
3.9 Breakdown of reasons for FABLE's inability to find aliases using different methods. 47
3.10 Breakdown of reasons why reliance on web archives to serve requests for these URLs would be undesirable. 50

5.1 No. of positives and negatives classified by each approach. 86

ABSTRACT

Web links are fundamental to the web, enabling navigation between pages and citations in research articles. However, web links suffer from "link rot", a phenomenon in which links are likely to become inaccessible over time. This can occur if a link's site disappears, making it impossible to resolve the hostname or establish a server connection, or the linked page has been deleted, resulting in an HTTP 404 "Not Found" error.

Today, a common solution to tackle link rot is to rely on web archives, which capture snapshots of web pages for future reference. However, the modern web has evolved significantly since web archives were first introduced, leading to several limitations in their effectiveness. First, the scale of the web has grown tremendously, making it infeasible to crawl every page whenever it changes. As a result, many broken links either have no archived copies, or the archived copies have stale content. Second, modern web pages rely heavily on increasingly complex and diverse JavaScript. This shift has made it more challenging for preserving fidelity in archived copies, significantly increasing both the computational cost of operating browser-based crawlers and engineering effort required to maintain accurate replay systems.

This thesis presents a set of solutions to cope with link rot on the modern web. My work aims to mitigate the various limitations of web archives. First, for broken links without any archived copy, or if the archived copy includes stale content or unavailable functionalities, I built Fable. Fable revives the dead link with the new URL to the same page whenever available. Compared to prior approaches, Fable revives 4.6K broken links—a 50% increase—with much higher accuracy. Second, for pages that require dynamic crawling, I show how web archives can achieve a better tradeoff between efficiency and fidelity. By carefully choosing 8.9% of pages to crawl dynamically and strategically reusing resources from those crawls, an archive can serve 99% of the remaining statically crawled pages without any fidelity loss. Third, for fidelity violations in archived copies that are caused by incorrect edits to crawled scripts, I built FidEx. FidEx reliably detects when an archived page differs from its original version and pinpoints the root cause. After fixing the most common errors pinpointed by FidEx, I reduced the fraction of pages for which FidEx reports a violation of fidelity from 15% to 9%.

CHAPTER 1

Introduction

Web links are the key building blocks of the web, providing the connective tissue that enables navigation and interconnection across pages. When one page references another, it does so through links, making seamless traversal between pages possible. Beyond navigation, links serve as citations in research articles and as signals of importance in search engine rankings, where the authority of a page is influenced by the number and quality of incoming links.

Yet, links are ephemeral. Over time, many become inaccessible–a phenomenon widely known as *link rot*. Links can break for various reasons: domains may expire (DNS errors), servers may go offline (connection errors), or the referenced resources may be removed. Consequently, users often encounter broken links in everyday browsing. Prior studies estimate the half-life of URLs to be only about two years [82, 13], and our measurements indicate that 17-29% of links on the web are no longer functional.

1.1 Web archives: Today's solution for link rot and their limitations

To mitigate the effects of link rot, web archives are used to preserve records of web pages for long-term access. Well-known archival services include the Wayback Machine [32], Library of Congress [7], Perma.cc [38], and UK Web Archive [14]. These archives serve two primary functions: 1) they periodically crawl the live web to capture snapshots of pages, and 2) they

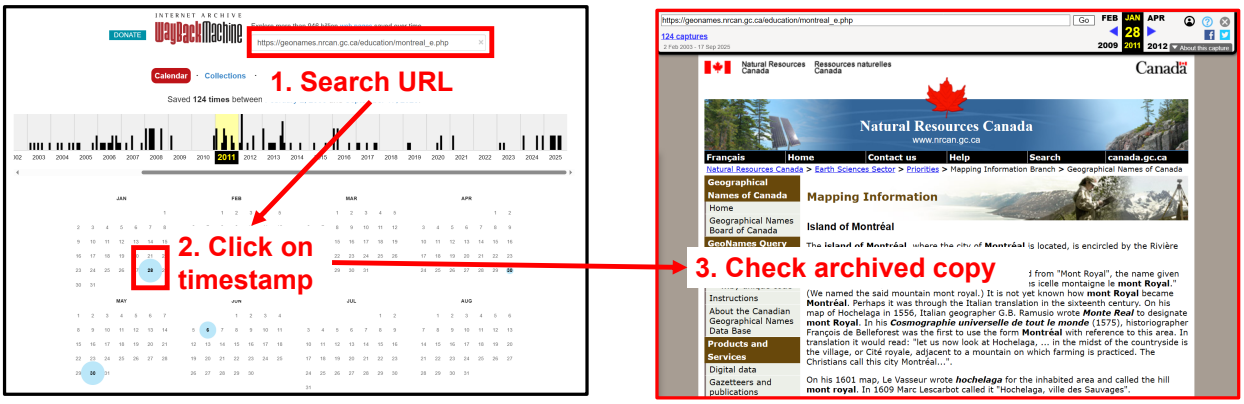


Figure 1.1: **Example of the Wayback Machine's usage.**

host these archived copies on their servers, enabling users to revisit and interact with pages as they appeared and functioned at the time of capture.

For example, as shown in Figure 1.1, when a URL is no longer accessible, users can turn to the Wayback Machine to search for its archived copies. If snapshots are available, users can select one to view how the page appeared and functioned at the time of capture. Moreover, users can continue navigating through archived versions by following outgoing links on the page, provided that the Wayback Machine has also archived the linked pages.

Web archives have been widely applied when users encounter a broken link. For example, the Internet Archive augments any broken reference on Wikipedia with a link to an archived copy of the referenced URL [33]. Similarly, the Brave web browser [24] and the Cloudflare CDN [25] direct any user who encounters a broken link to an archived copy of the page at that URL.

Although web archives offer valuable snapshots of pages when live links become inaccessible, they face important limitations. These challenges arise at multiple stages of the archival process:

- **Archived copies are unavailable, fail to provide functionality, or include stale content.** Archived copies often fall short as true replacements for their live web counterparts for several reasons. First, since it is impractical to archive the entire web,

even the Internet Archive – the largest web archive in operation today – has no copy for many pages [64, 65, 117, 30]. Second, modern web pages typically include functionality that requires client browsers to interact with backend servers, e.g., to make a purchase or post a review on a product page; such functionality will not work on an archived snapshot of a page [85]. Lastly, since page content is often updated over time, the last archived copy of a page may be stale, e.g., after a news article was last archived, some important corrections might have been posted [83].

- **Crawling is either too expensive, or prone to missing resources and leading to low-fidelity for archived copies.** Traditional archival crawlers record pages statically by downloading the HTML, parsing it to extract links, and recursively fetching the linked resources. This *static* crawler, however, fails to capture resources that are dynamically loaded, such as those fetched via JavaScript (e.g., AJAX requests). Missing such resources often results in archived pages that fail to accurately reproduce the appearances or functionalities of the original.

  To address this limitation, newer browser-based archival crawlers have been introduced [48]. By executing JavaScript during the crawl, they are able to capture dynamically fetched resources and produce higher-fidelity copies. While effective, browser-based crawling incurs significant computational overhead, largely reducing crawling throughput. Given the vast scale of the web, archives cannot feasibly rely on browser-based crawlers for every page.

- **Serving incorrectly rewrites resources, causing fidelity violations in archived copies.** Web archives aim to serve archived copies with high-fidelity: i.e., when a user loads an archived copy of a page in a browser, it must closely match the original page visually (i.e., content and layout) and functionally (e.g., menus, buttons). The key to achieving this goal is to modify crawled pages before serving them to users, so that the user's browser fetches all resources from the archive, not from the original domains.

In the past, it sufficed for archives to rewrite resource URLs which are statically embedded in a page's source to point to the copies of these resources stored by the archive. However, on modern web pages, an archive must also instrument scripts so that resource URLs which are dynamically constructed get rewritten at runtime [17, 19]. Unfortunately, due to the wide variety of JavaScript code prevalent across the web, an archive's edits to crawled scripts often affect their execution in unanticipated ways. As a result, with existing systems for serving an archived copy of a page, either some content present in the original page is missing or some functionality on the page is broken [11].

1.2 Thesis and Contributions

In this thesis, I present a series of solutions that address the problem of link rot on the modern web. In doing so, this dissertation supports the following **thesis statement**: *When users follow broken links, to maximize the fidelity of the page served to them, we must account for how pages are hosted by administrators, composed by developers, and rendered in browsers.*

My work spans three key aspects of the process, from when a page is served to when it is rendered in users' browsers. Specifically, I (1) developed a system that revives dead links by finding the new links of the same pages hosted on the live web, (2) conducted a study to improve the efficiency and fidelity of web archival crawling, leveraging the observation that pages are typically composed with reusable resources across time and within the same site, and (3) built a system that helps archival developers detect and diagnose incorrect serving of archived copies, by understanding how pages are rendered in browsers.

Reviving dead links with FABLE**.** I begin by resolving the first limitation of web archives when archived copies fail to serve as a good replacement compared to their live counterparts. The key observation enabling us to derive a solution that helps for these cases is that: a broken link to a page does not always mean that the page no longer exists on the

web. Rather, the link may be broken only because the page's URL has changed as a result of a reorganization of its site hosting it. In such cases, if a user visits the linked page using its new URL (which we refer to as an *alias* of the page's original URL), the user can access up-to-date page content and use all services on the page.

To automate the process of finding aliases given broken links, I built FABLE. The key insight for designing FABLE is that when a site is reorganized, we observe the URLs of many pages change, and the new URLs are derived in a similar manner from their corresponding old URLs. FABLE unearths the aliases for a set of similar broken URLs (e.g., URLs in the same directory) by looking for a common URL transformation pattern from these URLs to the URLs of pages that currently exist on that site.

I performed a large-scale study of the prevalence of new URLs for broken links on the web, by using FABLE to identify aliases for 20K broken external links found on the pages of Wikipedia, Stack Overflow, and Medium. We estimate that the fraction of these linked pages for which FABLE correctly found the new URL to be 1.5x that feasible with prior approaches that solely rely on similarity in page content.

Toward Better Efficiency vs. Fidelity Tradeoffs in Web Archives. The second limitation stems from the tradeoff between efficiency and fidelity: static crawlers are lightweight but often produce low-fidelity copies, while dynamic crawlers achieve high fidelity but are too resource-intensive to run at scale. To address this, I show that web archives do not have to choose between the efficiency of static crawlers and the fidelity offered by dynamic crawlers. Instead, they can get the best of both worlds by using a browser for a small subset of crawls, which can then help "fill in" the resources missing in other static crawls.

My study demonstrates three key insights. First, simply reusing crawled resources across archived page snapshots can significantly make up for the incompleteness of static crawls. Missing resources in one crawl are often available from another dynamic crawl of the same page—even one conducted 10 weeks earlier or later—or from a dynamic crawl of a different page on the same site. Second, when replacing missing scripts, fidelity often requires

replacing them collectively rather than in isolation. Third, for missing resources that are unique to a page and vary over time, I show how to infer the recipe for constructing such resource URLs from a single dynamically crawled copy of a page. A static crawler can then be augmented to fetch these resources which it would otherwise miss.

To evaluate the utility of our proposed methods, I repetitively crawl tens of thousands of diverse web pages over the span of 16 weeks. Put together, our results suggest that, if an archive recrawls pages at the same frequency as we see on the Wayback Machine, then using a browser for about 8.5% of its crawls suffices. The methods I present will then enable the archive to serve over 98% of the remaining statically crawled page snapshots at high fidelity.

Detecting and Diagnosing Errors in Serving Archived Web. The final limitation arises from fidelity violations in archived copies caused by the necessary–but often incorrect–edits that archival systems make when serving pages. To ensure fidelity, developers must repeatedly refine these edits. However, doing so requires both the ability to detect fidelity violations introduced by erroneous rewrites and to diagnose their underlying causes.

To more reliably detect fidelity violations and to pinpoint the cause when this occurs, I developed FidEx. FidEx monitors the execution of scripts both when a page is crawled and when its copy is later loaded. The information that FidEx collects enables it to accurately identify when differences between loads of the original page and its archived copy are indicative of user-visible problems. FidEx's fine-grained visibility into the execution of scripts also enables it to accurately hone in on the erroneous rewrites that result in poor fidelity, thereby providing the developers of archival systems valuable input to debug and fix the problem.

I have used FidEx to check the fidelity offered by pywb [17] on diverse collection of archived pages: US government web pages that were archived during the presidential transition in 2016 and 2020, pages archived from art websites, and random pages drawn from a wide range of site rankings. When FidEx identifies a fidelity violation, I find it to be wrong on less than 10% of pages and it is able to pinpoint the root cause on 64% of pages.

Importantly, I worked with the developers of pywb and fixed a handful of bugs which were responsible for the incorrect rewrites identified by FidEx on a majority of affected pages. By doing so, I reduced the fraction of pages for which FidEx reports a violation of fidelity from 15% to 9%.

1.3 Organization

The dissertation is organized as follows: In chapter 2, I describe the types of link rot and their reasons, and how web archives operate to provide archived copies for broken links. I also explore the prior work that aims to tackle link rot or can be used to improve web archiving. In chapter 3, I describe the design of Fable, a system that enables dead link revival by finding their aliases accurately and efficiently. In chapter 4, I describe the large-scale study I performed to explore better trade-offs between fidelity vs. efficiency in web archival recording through strategic crawling and resource reusing. In chapter 5, I describe the design FidEx, a system that helps improve web archiving's replay, by detecting fidelity violations on archived copies accurately, and diagnose the root causes of incorrect rewrites to the pages' JavaScript. I conclude the dissertation and discuss potential future work in chapter 6.

CHAPTER 2

Background and Related Work

2.1 Background

I begin by presenting the prevalence of link rot on the modern web, explaining how web archives are operated to provide archived copies for broken links, and discussing their limitations for preserving pages on the modern web.

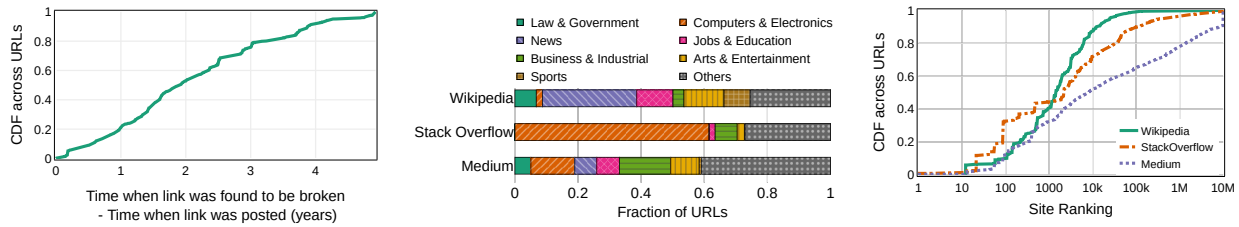
2.1.1 Prevalence of link rot

To quantify how frequently links break, I inspect the status of external links included on three sites: Wikipedia, Medium, and Stack Overflow. I select these as examples of sites whose pages 1) will be of value even several years after they are posted, 2) are not behind paywalls, and 3) often contain links to pages on other sites. On Wikipedia, I randomly sample the articles in which the Internet Archive has found at least one broken external link [29]. For Stack Overflow and Medium, I sample from all pages on these two sites using a data dump from Stack Exchange [42] and the sitemap for Medium [36].

For every link that I crawl, I declare the link as broken for one of three reasons: 1) I was unable to issue an HTTP request (e.g., DNS failure, TCP/TLS connection timeout), 2) I received an HTTP 404 error response, or 3) I deem the link to be a soft-404. For the last of these, I declare a URL u as broken if requests for it and for another randomly generated (and hence, invalid) URL u' redirect to the same target, provided it is not the site's login

Site	#Pages	#Unique links	#Broken links (%)
Wikipedia	40,000	1,024,435	297,594 (29.0%)
Medium	188,051	393,636	66,014 (16.8%)
Stack Overflow	265,027	161,454	30,920 (19.2%)

Table 2.1: **Sizeable fraction of external links are broken:** On three sites, number of pages crawled, number of unique external links seen, and the number and fraction of those links which are broken today.



(a) (b) (c)

Figure 2.1: **Links break a few years after they are posted, and broken links point to a wide variety of external sites:** (a) Among the external links posted to Wikipedia since 2017, for 2000 randomly sampled links which are dead now, time gap between when link was created and when the InternetArchiveBot [33] first found the link to be broken. Distribution across broken URLs along two dimensions: (b) category and (c) Alexa ranking of the domains to which these URLs belong.

page; I generate u' by replacing the suffix in u following the last occurrence of '/' with a random string of 25 characters [66].[1] While my detection of soft-404s misses some broken URLs (e.g., my method does not work when the URL to a site's homepage is broken, and it misses erroneous 200 status code responses [133]), I ensure that I do not classify a working URL as broken.

As seen in Table 2.1, 17–29% of external links are broken. Moreover, for every broken link in a Wikipedia article, I can identify the time gap between when the link was first added to the article and when it was marked as broken by the InternetArchiveBot [33]; I do so by using the article's edit history to inspect all versions of the article. I see that most broken links worked for only a few years after they were first created; Figure 2.1(a) shows that the

[1] Additionally, 1) for URLs with a numeric string in the middle (e.g., article IDs in the URLs for news articles), I also consider u' where I replace this token since this string may dictate the server's response, and 2) if the response to u contains a canonical URL [28] within the HTML, I find that almost always indicates a non-erroneous response.

median broken link became dysfunctional less than two years after it was posted.[2] These numbers do not bode well for the long-term sustainability of pointers to external information and services.

Though I focus on broken links found on the pages of three sites, the URLs corresponding to these links are from a wide range of domains, both with respect to site category and site popularity. I use the Public Suffix List [40] to map any URL's hostname to its domain, and query Klazify [35] and Alexa [27] to map any domain to a category and its ranking. Figures 2.1(a) and 2.1(b) show that there are some differences across the three sites I crawled, e.g., broken URLs found on Stack Overflow are predominantly from "Computers & Electronics" sites, whereas pages on Medium link to more broken URLs from lower-ranked domains. But, in all three cases, the broken external URLs point to pages from a wide variety of sites.

Note that, although my dataset only includes external links from user-generated content platforms (because, as mentioned above, these sites do not impose any limits on how many pages can be crawled), broken links are a pervasive issue, affecting a wide range of websites. For example, prior studies have observed that approximately 25% of the external links in articles published by The New York Times have become inaccessible [145], and at least 23% of the links included in the citations from research articles are invalid [103]. This rate of link rot aligns closely with my own findings across three different websites.

2.1.2 Web archival

Operations of web archives. Web archives are essential for preserving information on the web when its links are no longer accessible. Popular web archives such as Wayback Machine crawl pages when they are still alive, and serve them as archived copies for users to visit. When a web archive crawls a page from the web, it captures all resources (e.g.,

[2]Given the overheads involved in examining all versions of every article without exceeding Wikipedia's crawling limit, I restrict this analysis to a random sample of 2000 links from all the broken external links I found on Wikipedia. I find that the distribution in Figure 2.1(a) is similar for different randomly chosen subsets of 500 from these 2000 links.

HTML, CSS, scripts, and images) on the page. These resources are typically stored in the Web ARChive (WARC) format [15]. An archive may collect many snapshots of the same page over time, in order to maintain a record of changes made to that page.

When a user wishes to view an archived page, they specify the URL of that page and choose the particular snapshot of interest. For example, a user can visit the URL https://web.archive.org/web/20201104102000/https://www.nytimes.com/ on the Wayback Machine to view a copy of the NYTimes home page captured by the Internet Archive on the morning of 4 November, 2020, the day after the US presidential election. To ensure that the user's browser fetches all of the page's resources from the archive and not from the original servers that hosted them, a web archive can employ one of two methods.

Static crawling vs. Dynamic crawling. Web archives have historically used so-called *static* crawlers to crawl web pages, A static crawler crawls a page by first fetching the page's HTML and parsing the HTML to extract links to other resources on the page, such as images, stylesheets, and scripts. It then fetches those resources and recursively repeats the process until all of the page's resources have been fetched.

However, static crawlers are increasingly proving to be insufficient for crawling pages from the modern web. Today, on many pages, the URLs of some resources are not included in the page's source; rather, these URLs are determined only when scripts on the page are executed. Such resources go unfetched by static crawlers. As a result, for over 45% of pages, when I load a statically crawled copy in a browser, I find that either important content is missing (see Figure 2.2 for an example) or functionality that ought to work on an archived copy (e.g., menu dropdowns) does not.

To address this problem, there has been a recent shift toward *dynamic* crawlers, which crawl every page by loading it in a web browser such as Chrome. As in the example in Figure 2.2, these crawlers enable an archive to serve pages without loss of fidelity.

Need for rewriting. One approach is to have the user's browser load any page from the archive via a trusted proxy, similar to many web record and replay tools [71, 115]. The

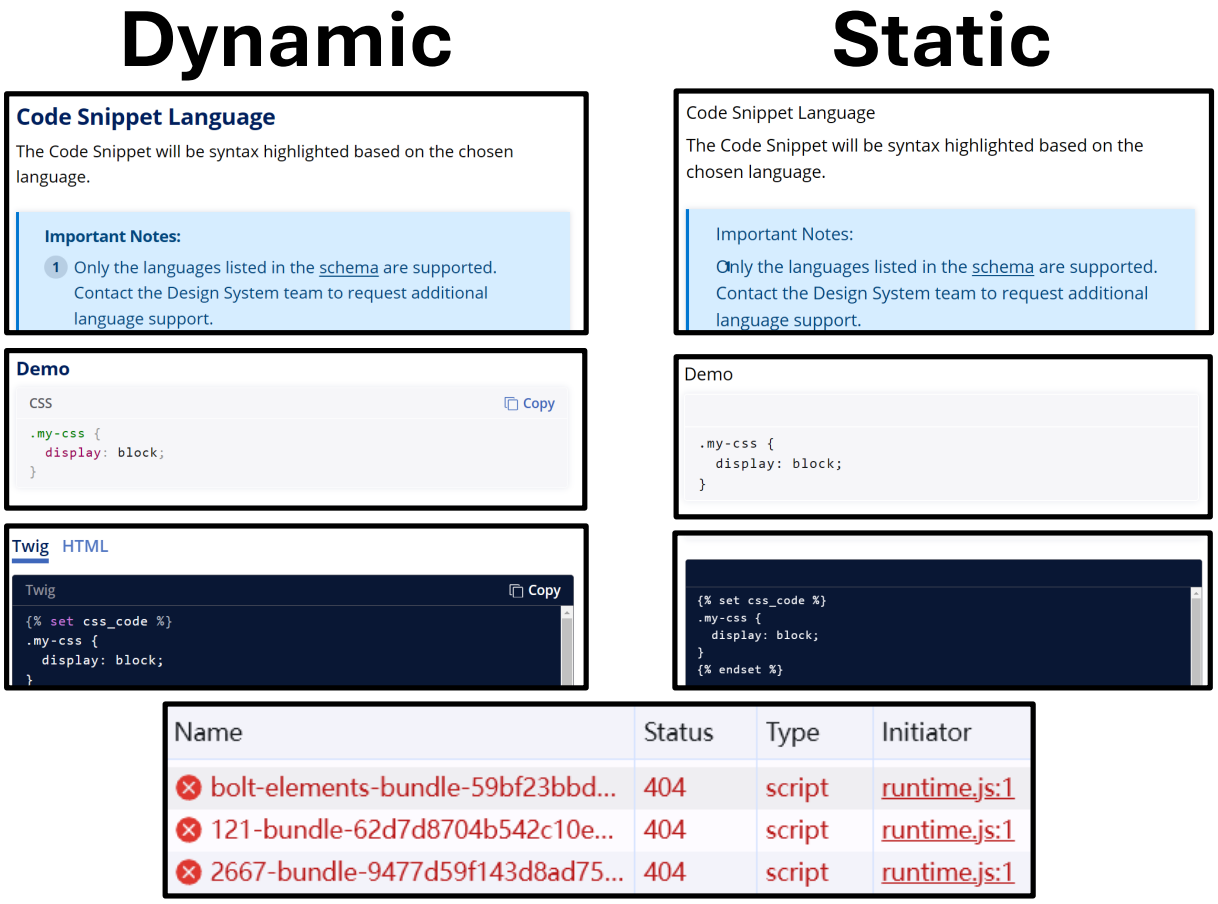


Figure 2.2: **Screenshots when loading copies of an example web page [46].** The dynamically crawled (left) and statically crawled (right) copies differ because of resources that the latter is missing (bottom). Wayback Machine's snapshot of the page [47] violates fidelity in an identical manner.

browser can request any resource using its original URL. The proxy relays the request to the archive after rewriting the URL on-the-fly to refer to the copy of the resource hosted on the archive. This approach enables the archive to serve crawled resources unmodified.

However, legacy browsers do not support site-specific proxy configurations. Therefore, if a user does not trust the proxy to handle *all* of their web traffic, consider what the user needs to do when they encounter a link on the web which leads to a "Page not found" error. To lookup an archived copy of the page at that link, the user will have to modify their browser configuration to use the proxy, access the archived page, and then switch back to not using

```
// Override fetch method at start of page load
let originalFetch = fetch;
fetch = function (url) {
    Code to change url to archived format ...
    return originalFetch(url);
}
```

(a)

```
{
  // "wombatAssignFunction" returns an overridden document object
  let document = wombatAssignFunction("document");
  (async function navigate(){
      let response = await fetch("/new-url.json");
      response = await response.json();
      document.location.href = response.url;
  })()
}
```

(b)

Figure 2.3: **Example of dynamic rewriting of resource URLs.** Code inserted by a system like pywb [17] is highlighted in green background. Objects and methods that are overridden are highlighted in blue bold text.

the proxy before continuing to browse the web. Requiring the average user to modify their browser configuration whenever they wish to access an archived page is impractical.

Therefore, most popular web archives such as Internet Archive's Wayback Machine and Library of Congress instead rewrite crawled resources. Web archives employ two kinds of rewriting techniques.

Static rewriting. Traditionally, web archives have rewritten all elements within a page's source that could result in a request to another resource. Common targets include attributes such as the `src` in `<img>` tags, hardcoded URLs in JavaScript's fetch requests, and URLs defined using the `URL()` function in CSS. Such modifications are made before resources are served to clients.

Dynamic rewriting. With the emergence of dynamically fetched content via JavaScript (JS), static rewriting is insufficient to account for resource URLs which are determined at

runtime. Hence, web archives now also perform dynamic client-side rewriting, as illustrated in Figure 2.3.

- First, the archive includes a script such as `wombat.js` [19] in the page's main HTML. This script, which executes before any other scripts on the page, overrides all JS functions and element methods that are capable of initiating requests. For example, in Figure 2.3(a), the `fetch` method is overridden so that, before an AJAX request for a resource is initiated, its URL can be modified to the corresponding archived copy.
- Some DOM native objects such as `window` or `document` cannot be overridden. To tackle this, web archives wrap the code in every JS file within a local scope (i.e., between "{}") so that all global DOM objects are changed into local objects, which can then be overridden. Additionally, a code snippet is inserted at the beginning of each script to facilitate access to the 'localized' versions of these objects. For example, in Figure 2.3(b), the setter of `document.location.href` is overridden so that its value will be prefixed with the web archive's hostname.

2.1.3 Limitations of web archives today

Today, the modern web has evolved significantly since web archives are first introduced. The change has brought multiple limitations for web archives to cope with link rot on today's pages.

No archived copy. The scale of the web has grown tremendously in the last 10 years. With the budgetary constraints, web archives do not have any copies for many URLs [64, 65] or attempt to archive a URL only once it no longer works [117]. In a random sample of 500 broken URLs from Table 2.1, I find 143 (28%) of these URLs have no archived copies on the Internet Archive's Wayback Machine.

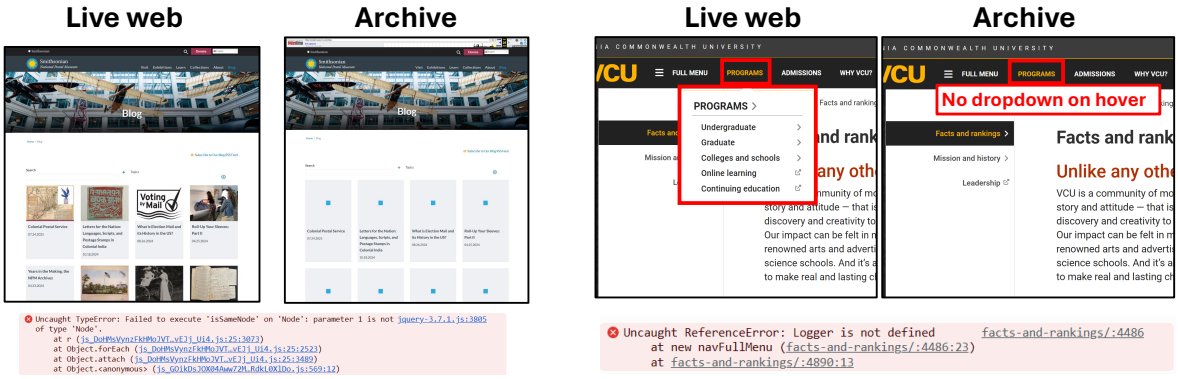
Archived copies have stale content. Many links on the web are either navigational (i.e., to point users to whatever content happens to be at the linked page when they follow the link) or transactional (i.e., help users discover some app-like functionality) [68]. For

such links, users have to make do with potentially stale content in the archived copies. To demonstrate, I take a look at the remaining 357 URLs from the sample 500 broken URLs that have archived copies. Out of the 357 URLs, 269 have multiple archived copies. For 29% of these 269 URLs, the core content of the page (i.e., parts of the page that remain after filtering out boilerplate [101, 31] such as sidebars, recommendations, etc.) differs significantly between the earliest and latest copy; based on prior work [119], I consider a TF-IDF similarity [127] of less than 0.8 as evidence that the page has been significantly modified. For these pages which are updated over time, even the last archived copy is likely to have stale content.

Archived copies lack functionalities. Modern pages are relying more on client-server interactions (e.g., comments, purchases). Archived copies cannot preserve these functionalities if the server is no longer available. My manual analysis reveals that the fraction of pages which include functionality that requires client-server interactions increased from 29% before 2010 to 69% after 2015. This suggests that functionality that will not work on archived copies has become more prevalent over time.

Limited budgets for dynamic crawl. Although dynamic crawl can preserve pages' fidelity much better, this benefit comes with the downside that the computational overheads of crawling pages with a browser makes the use of dynamic crawlers impractical at scale. Recent work [87] has reported a 8x drop in crawling throughput with dynamic browser-based crawlers, as compared to static browserless crawlers. Since web archives are typically run by cash-strapped non-profit organizations, scaling up their infrastructure to support the use of dynamic crawlers is not an option. Therefore, today, archives largely use static crawlers and make do with the corresponding loss of fidelity.

For example, I estimate that only around 5% of the archived pages available on the Wayback Machine were crawled with a browser. The estimation was as follows: For any given page snapshot on the Wayback Machine, I determine whether it was statically or dynamically crawled based on the differences in the times at which the resources in that snapshot were captured. In static crawling, any page resource (e.g., script or images) discovered when



(a) (b)

Figure 2.4: **Screenshots when loading pages from** (a) postalmuseum.si.edu and (b) vcu.edu from the web (left) and from an archived copy (right). In (a), images for blog posts are missing even though they were crawled. In (b), hovering on the menu bar fails to reveal the dropdown. At the bottom, I show the cause: the runtime errors that result from the archival system's rewriting of scripts.

parsing the HTML is appended to a queue. A resource is fetched when it later arrives at the head of the queue, often several minutes later at high crawling throughput. In contrast, dynamic crawlers load pages in a browser, which fetches resources immediately after the HTML is received and parsed. I classify a snapshot as static if no resource is captured within 10 seconds of the HTML; otherwise, I treat it as coming from a dynamic crawler.

Fidelity issues due to dynamic rewriting. Given the vast diversity of JS code on the web, rewriting every page's source as described above often leads to a variety of unexpected errors. During a user's load of an archived page, a syntax error or runtime error introduced by overriding JS objects and methods either prevents or halts the browser's execution of the script in which this error occurs. Even if there are no overt errors, the archive's rewrite of scripts can alter the flow of execution, e.g., by causing the values of variables to diverge. As a result, any content/functionality for which the code that does not get executed was responsible is rendered missing/broken.

To show that this problem is not specific to any one archival system, Figure 2.4 presents two examples where the archived copy of a page fails to mimic the original page: the first example is of a page served by the Wayback Machine [3], and the second was replayed with

pywb [17]; the first page also exhibits the same issue with ReplayWeb.page [18], another mature open-source framework for loading archived pages. In all cases, client-side dynamic rewriting results in unexpected runtime errors, which preempt the execution of JS code responsible for constructing the rest of the page.

Consider the example in Figure 2.4(a). On this page, images are lazily loaded by a script which modifies the `src` attribute of every image after the page's `load` event fires. On any load of the Wayback Machine's copy of this page, the event handler which runs this script fails to get registered because of an error introduced by the archival system's override of the `document` object to rewrite its APIs. Specifically, due to incorrect rewrite, the overridden and native versions of the `document`'s `documentElement` attribute are compared, which results in an uncaught exception because they are objects of different types. This runtime error prematurely halts the execution of the script which registers the above-mentioned event handler.

2.2 Related work

This section provides an overview of prior work in the related areas. Specifically, I describe studies on broken links as well as alternatives to web archives, crawling and caching techniques that can be used by web archives, and testing and debugging methods for general programs.

2.2.1 Broken links

Detecting and studying broken URLs. Many tools and algorithms have been developed to detect web decay. Bar-Yossef et al.'s approach [66] detects soft-404 pages by comparing the request behavior to the URL with a constructed URL that does not exist. Parking Sensors [133] identifies unused domains which only include automatically computed advertisements and links.

Prior work has also extensively studied the evolution of the web graph and web pages, both with respect to the persistence of links [66, 103, 130, 100] and stability of page content [100, 116, 74, 82]. My observations are in line with these studies; I too find that a sizeable portion of URLs from the past are now invalid. These prior efforts have also observed that flux in page content is common, which is why it is often insufficient to serve archived copies in response to requests for broken URLs.

Rediscovering web pages. Many others have previously presented solutions to fix URLs that break when websites are reorganized [96, 98, 99, 83]. However, they largely rely on using a "lexical signature" extracted from a page's content as a robust hyperlink [120, 119], an approach which results in poor coverage, accuracy, and efficiency, as I will show in Chapter 3. Others, similar to me, have observed the potential to leverage similarity across pages on a site in how their URLs are transformed when the site is reorganized [83]. They have not designed or evaluated a system which exploits this observation.

Duplicate page detection. Detection of a web page's duplicates has been widely researched [81, 90, 111, 131] in order to identify plagiarism and phishing, save storage by de-duplication, etc. Applying prior techniques from this line of work to find duplicates of archived copies is insufficient to find aliases for broken URLs because not all URLs are archived and archived copies can be stale.

2.2.2 Web crawling and caching

Web crawlers have been extensively studied to improve their efficiency [75, 74, 91] and their ability to accurately crawl pages of specific properties [73, 80, 72]. A range of open-source static [53] and dynamic [48, 52, 51] crawlers exist for web archiving. Prior work [87] has also proposed combining browser-based and browserless crawling to maximize crawling throughput without missing out on any resources fetched by a browser. However, unlike general crawling tasks, archiving web pages can have missing resources as long as they have no impact on the page's appearance and functionality. Moreover, many proposed novel web

crawlers have dedicated processes for crawling, causing a lot of necessary changes to how web archives currently operate.

Web caching. Prior studies have identified redundant network traffic despite the use of web caches due to client-side caching implementations that fail to adhere to protocol specifications [123, 139], or from variations in URLs and improper caching headers for cacheable content [114, 95, 107]. Solutions to improve cache utilization include content-based caching [112], cross-application cache sharing [139], and cache lifetime extensions for unchanged resources [124]. While web archives can also reuse of resources across page loads, their focus on archived pages has several differences compared to the loads of pages from the live web.

2.2.3 Program testing and debugging

Web application testing. There have been many studies to detect Cross-Browser Incompatibility (XBI), which involves comparing web application behavior across browsers. Solutions for XBI have employed both DOM and visual comparisons to detect discrepancies [78, 76], and also compared the relative layouts of web applications across different browsers [77]. An alternate, but related, line of work has studied HTML presentation failures: discrepancies between the actual rendering of a page and its intended appearance. Prior work has developed methods for the automatic detection and localization of HTML presentation failures using image processing and computer vision techniques [109, 108, 110].

Fault localization. To identify the root cause of buggy programs, prior work has largely used two types of approaches. One class analyzes crash stack traces or bug reports to localize problems [128, 144, 126, 140]. Another relies on tests to evaluate program behavior and flag suspicious code, using spectrum-based [93, 63], mutation-based [118, 138], or ML-based [105, 106, 136] fault localization. On the one hand, the former class of techniques usually requires the crash to halt the program completely, or make significant consequences to the execution so that the faulty code is directly related to the stack trace. On the other

hand, the latter set of methods often need enough number of test cases to expose behavioral diversity in different tests to accurately identify suspicious lines of code.

JavaScript record and replay. Recording and replaying JavaScripts has been widely used for debugging and performance evaluation [113, 129, 125, 71, 69]. In these systems, JavaScript code is not rewritten after crawling. Instead, the browser is configured to run in an environment wherein, even though the browser requests the page's resources using their original URLs, these requests are directed to the local copy. The focus is on helping developers debug by faithfully replaying the execution during recording. In contrast, web archives serve archived copies of pages after rewriting JavaScript; as I showed, this often affects their behavior.

CHAPTER 3

Reviving Dead Links on the Web with FABLE[1]

In this chapter, I focus on the limitations of web archives where archived copies fail to be available or serve as a good replacement compared to their live web counterparts. I observe that broken links are often merely a result of website reorganizations; the linked page still exists on the same site, albeit at a different URL. I describe FABLE, a new system that attempts to find the linked page's new URL by learning and exploiting the pattern in how the old URLs for other pages on the same site have transformed to their new URLs. I show that our approach is significantly more accurate and efficient than prior approaches which rely on stability in page content over time. FABLE increases the fraction of dead links for which the corresponding new URLs can be found by 50%, while reducing the median delay incurred in identifying the new URL for a broken link from over 40 seconds to less than 10 seconds.

3.1 Motivation

There have been several efforts to address broken links when web archives fail to provide suitable archived copies. However, each of these approaches has notable limitations.

Making URLs resilient. Users do not have to make do with archived page copies in cases where a link to a page is broken only because the page's URL has changed; see Table 3.1 for examples. To preempt such link breakages, one could rely on content-based

[1]The content of this chapter is revised from the previously published publication "Reviving Dead Links on the Web with Fable [142]"

Problem with archived copy	Example broken URL with functional alias
Stale content	***Broken URL:*** http://www.hks.harvard.edu/centers/mrcbg/about/fellows/currentsrfellows Set of senior fellows listed in the last archived copy [23] is out-of-date compared to content at page's new URL ***New URL for the same page:*** https://www.hks.harvard.edu/centers/mrcbg/about/senior-fellows
Inaccessible service	***Broken URL:*** http://www.sup.org/book.cgi?id=21682 One can buy the book on the live page, but not on an archived copy [22] ***New URL for the same page:*** https://www.sup.org/books/title/?id=21682
Missing copy	***Broken URL:*** http://elections.nytimes.com/2010/house/new-york/03 There is no non-erroneous archived copy of the page's original URL, but the page is still on the web ***New URL for the same page:*** https://www.nytimes.com/elections/2010/house/new-york/3.html

Table 3.1: **Examples of problems associated with using archived copies in cases where the page still exists at an alternate URL.**

addressing, such as IPFS [34], wherein pages are retrieved by their content hashes instead of their locations. However, such an approach works well only for static blobs of content, and is ill-suited for navigational links to pages whose content is updated over time.

Alternately, to serve requests for any page irrespective of whether it is requested using its old or new URL, the provider of that page can include a page-specific unique ID in the page's URL [102]. But, website providers need to then make sure to preserve every page's ID in its URL even after reorganization. I find that sites which include page IDs in their URLs often violate this requirement. For example, the page that was previously at http://www.technologyreview.com/article/419483/measure-for-measure/ is now at https://www.technologyreview.com/2010/06/22/202620/measure-for-measure/; the page-specific ID included in the page's URL has changed from *419483* to *202620*.

Rediscovery of missing pages. To sidestep the challenges associated with making URLs on the web resilient to site reorganizations, prior work [89, 97] has studied how to

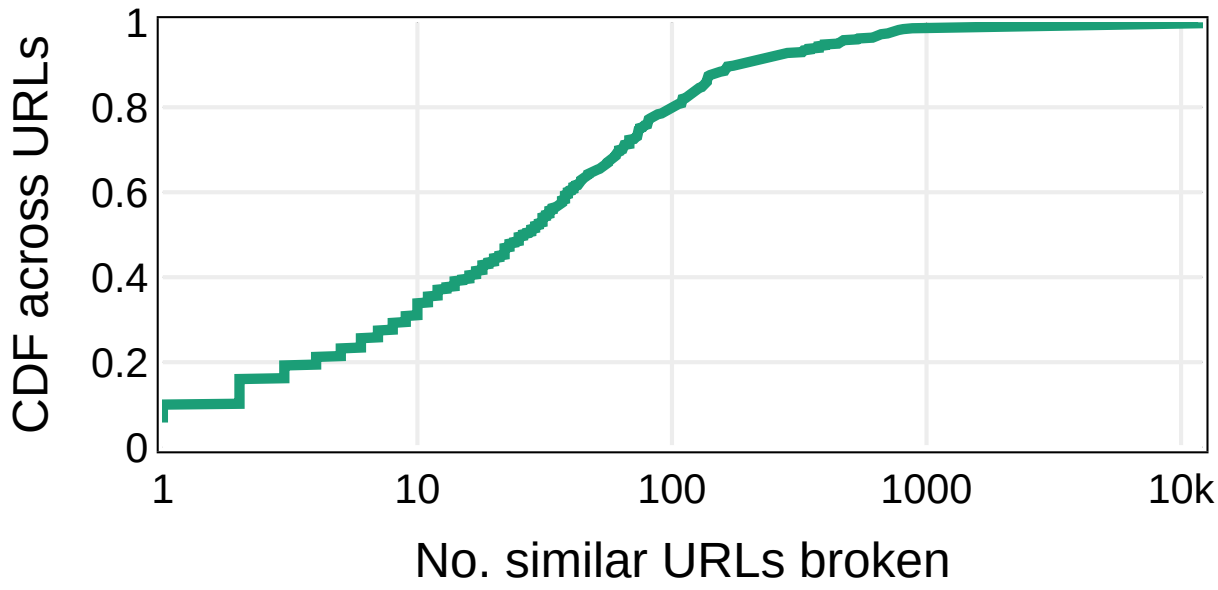


Figure 3.1: **Many URLs on a site go dead together:** For each of 500 broken URLs, number of similar URLs on the same site which also stopped functioning after the URL became dysfunctional. Note logscale on x-axis.

discover the new URL (i.e., alias) corresponding to any broken URL. Mimicking what a human user is likely to do, they retrieve a copy of the page that was previously available at the URL which is now broken, extract a variety of features from this copy (such as page title, lexical signature [120], etc.), and then use these features to query web search engines. Existing solutions simply show the search results to the user, and they let the user identify which result, if any, is a link to the same page. One could instead automate the process of checking if a search result is a close match to the page's stored copy.

I observe that such an approach for discovering a page's new URL is prone to several fundamental limitations.

- *Poor accuracy:* First, the archived copy of the broken URL and the page that currently exists at any of the search results might share similar features only because they happen to be two similar pages. For example, the archived copy of http://marvel.com/comic_books/issue/22962/what_if_2008_1 [21] has the title ("What If? (2008) #1"). Though the live page at https://www.marvel.com/comics/issue/22964/what_if_2008_1 has the same

title and similar content, it is a page for a different book.

- *Poor coverage:* Attempting to find a page that is similar to a broken URL's last archived copy will fail to find any match when no copies exist for that URL – which is common, as I showed earlier in section 2.1.3 – or when the page has been significantly modified since it was last archived.[2] To improve coverage, one could reduce the degree of similarity with the archived copy needed for a search result to be considered a match. But, this will further degrade accuracy.
- *Poor efficiency:* Finally, the workflow outlined above requires performing multiple web search queries and crawling tens of web pages to find the alias for each broken link. Doing so offline for all broken links across the entire web is not practical and will make online low-latency discovery of aliases infeasible.

3.2 Overview of FABLE

Guiding observations. I design FABLE to help users rediscover the live page once the old links to it break because of a change in the page's URL. To address the aforementioned limitations of existing solutions for doing so, FABLE's distinguishing characteristic is that it is not reliant on checking whether a page's content/title before its URL changed is similar to what is currently on the page. I observe that, when a URL ceases to work, it is usually the case that other similar URLs on that site (e.g., those in the same directory) also suffer a similar fate. Figure 3.1 demonstrates this property for 500 URLs chosen at random from those which have at least one successful and one erroneous archived copy on the Internet Archive. For the median URL u, I find that 26 other similar URLs (i.e., with the same prefix until the last '/' in u) were originally functional but stopped working after u ceased to work; they were successfully archived by the Internet Archive at least once but their archived

[2]Once a broken URL's alias is discovered, determining whether the page's content has sufficiently drifted that it no longer serves the purpose for which it had been linked to is a separate problem [145, 94] that is outside the scope of this work.

1	URL	cbc.ca/news/story/2000/01/28/pankiw000128.html
	Title	Pankiw will not be silenced
	Alias	cbc.ca/news/canada/pankiw-will-not-be-silenced-1.249577
2	URL	cbc.ca/news/story/2000/07/12/mb_120700Potter.html
	Title	Potter book flies off shelves
	Alias	cbc.ca/news/canada/potter-book-flies-off-shelves-1.201722
3	URL	cbc.ca/news/story/2000/07/04/rancher000724.html
	Title	Rancher survives tornado
	Alias	cbc.ca/news/canada/rancher-survives-tornado-1.215189

Table 3.2: **New URLs for the pages on a site are typically derived in the same manner from their old URLs:** Example of a group of 3 broken URLs, their titles and corresponding aliases.

copies after u stopped functioning are erroneous.

In addition, I posit that changes in page URLs are typically the result of programmatic reorganization of an entire site or subdomain, rather than manual movement of pages. Consequently, I observe that there is usually a pattern in how the old URLs for pages on a site map to their corresponding new URLs. Table 3.2 shows an example.

Workflow. To exploit the above observations, FABLE combines a frontend – either a browser add-on or a bot that rewrites links on pages – and a backend service; see Figure 3.2.

When a user who uses FABLE's browser add-on visits a broken URL, it gives the user the option of visiting either an archived copy of that URL or FABLE's predicted alias for the URL; Figure 3.3 shows the user interface. The frontend predicts the alias based on URL transformation patterns that it has previously received from the backend. Alternately, website providers can use FABLE's link rewriter in a manner similar to the InternetArchiveBot [33], which augments broken links on Wikipedia with links to archived page snapshots. FABLE's bot can scan all pages on a site, identify the external links that do not work, use previously identified URL transformation patterns to try to identify an alias for each such link, and rewrite the link to add the alias if one is identified.

In either incarnation of FABLE's frontend, I take care to ensure that the alias that it identifies for a broken link is presented as an alternative to, not a replacement for, the

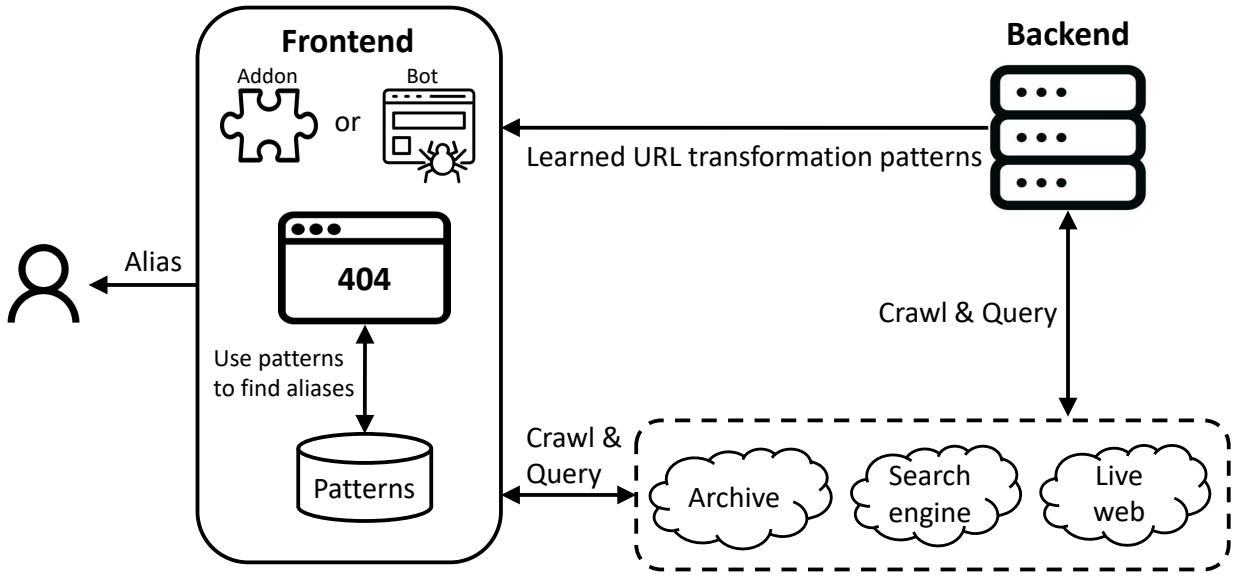


Figure 3.2: **Overview of FABLE's architecture.**

original link. Thus, when the identified alias turns out to be incorrect, the user still retains the option to visit the original link. Existing systems [38, 41] that augment web links with pointers to archived page copies similarly give users the option to visit either the live page or the archived copy. Moreover, for any broken URL, FABLE restricts its attempt to find the alias to an alternate URL on the same site. Therefore, even if the URL transformation pattern shared by the backend is incorrect (either because it is buggy or compromised) or FABLE's predicted alias turns out to be wrong, the user is trusting the same site provider when visiting the predicted alias as when the user chooses to follow the original link.

Both the add-on and the page rewriting bot rely on URL transformation patterns discovered by FABLE's backend. For this, the backend first groups together all broken URLs that are in the same directory on a site. It then considers one group at a time and attempts to find aliases for all URLs that it is aware of in that group. In doing so, the backend learns how these URLs were transformed into their corresponding new URLs.

In my current implementation, the backend discovers broken URLs simply by crawling the public web. To better scale FABLE and to help the backend discover more broken links

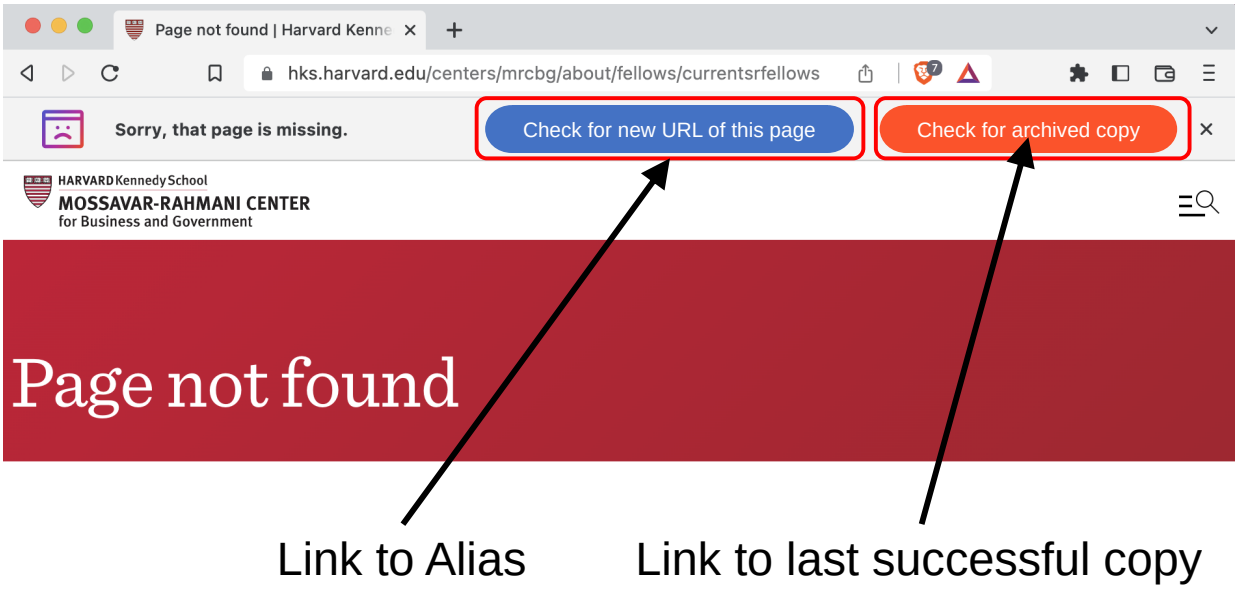


Figure 3.3: **When a user who has installed FABLE's browser add-on visits a link that returns an error, the add-on provides links to the latest archived copy (if one exists) and to the link's functional alias (if one is identified).**

on the web (e.g., links on pages that are not publicly accessible), an alternative design could have frontends share with the backend broken URLs that they are unable to map to aliases. However, this would have to be done with care, as the URLs shared by a frontend may potentially reveal sensitive information. Designing a privacy-preserving mechanism for this purpose is beyond the scope of this work.

Goals. In designing FABLE, I am guided by three goals, in the following order of priority.

- **Accuracy.** Of the aliases found by FABLE, I strive to maximize the fraction which are correct. If many of the identified aliases are suspect, users and website providers are less likely to use it.
- **Coverage.** I seek to find aliases for as many input URLs as feasible. Ideally, if the page that a URL was previously pointing to still exists, FABLE should be able to find the page's new URL.
- **Efficiency.** I aim to minimize 1) the work (number of pages crawled, search queries

Component	Goal	Technique	Section
Backend	Reduce volume of web search queries	Find aliases by identifying non-erroneous historical redirections	§3.3.1.1
	Minimize need to crawl pages from the live web Enable better accuracy than matching based on similarity in page content	Match broken URLs to search results based on URL transformation patterns	§3.3.1.2
Frontend	Minimize need to crawl or query any external service Find aliases for URLs with no archived copies	Infer aliases based on the transformation patterns learned by backend	§3.3.2.1
		Skip URLs for pages that are likely deleted	§3.3.2.2

Table 3.3: **Summary of the techniques used in FABLE to maximize accuracy, coverage, and efficiency.**

issued, etc.) that FABLE's backend must do to locate aliases for dead links, and 2) the delay between when a user encounters a broken link and when FABLE is ready to direct the user to the alias.

3.3 Design

I describe my design of the FABLE backend and frontend by answering three questions.

- Given a set of similar broken URLs, how can the FABLE backend find aliases for all of them, if the pages that those URLs previously pointed to still exist?
- In what form should the FABLE backend represent the URL transformation patterns that it learns?
- How should a FABLE frontend use these patterns to discover the alias for any dysfunctional URL?

My reliance on the consistency in how old page URLs on a site map to the corresponding new URLs helps maximize FABLE's coverage and accuracy. Whereas, I make FABLE efficient by 1) reducing the need for the backend to query web search engines and crawl pages from the

web, and 2) maximizing the frontend's ability to find aliases locally. Table 3.3 summarizes the key techniques.

3.3.1 Backend design

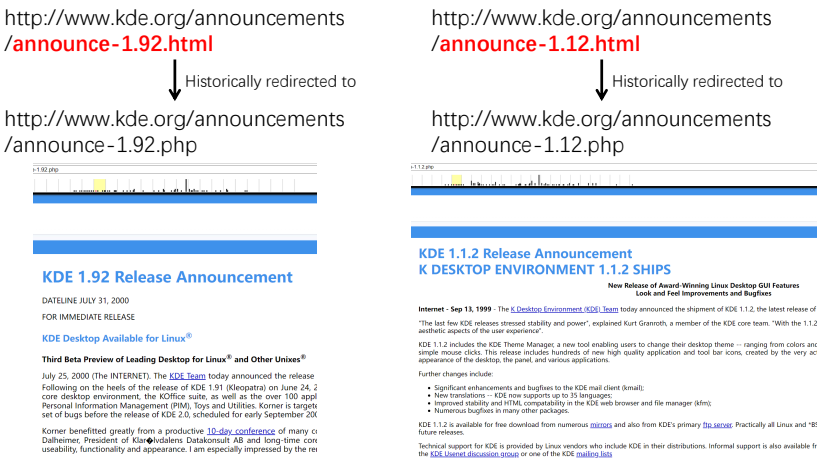
For all the broken URLs that the FABLE backend collects – either by crawling the web (like in my current implementation) or via uploads from frontends – it attempts to discover the corresponding aliases using two methods. The primary goal here is to learn the patterns underlying how these URLs have been transformed so that, when a frontend encounters different but similar broken URLs, it can use these patterns to locally identify their aliases.

3.3.1.1 Leverage historical redirections

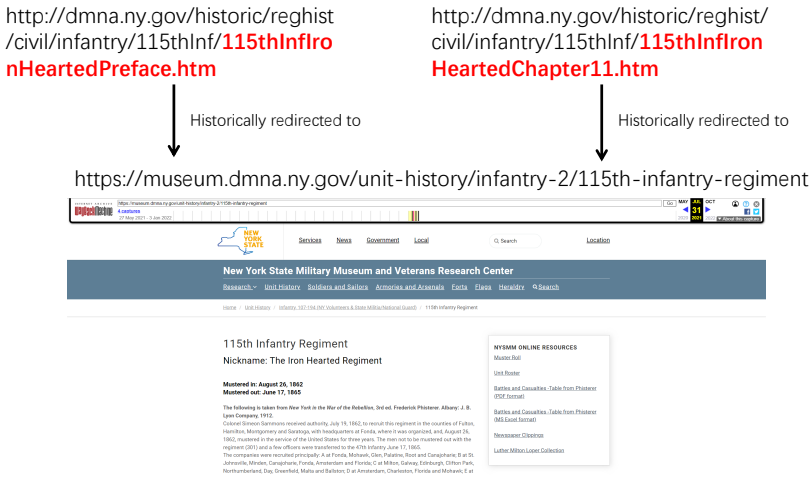
Approach and challenge. The high-level observation that guides my first approach is that many URLs that no longer work today did previously redirect to the corresponding aliases. In other words, after reorganization, some sites initially redirect requests for any page's old URL to the new URL for that page. But, these sites subsequently lose the state necessary to perform such redirections. For example, the URL http://www.kde.org/announcements/announce-1.92.htm does not work today, but it previously redirected to its alias http://www.kde.org/announcements/announce-1.92.php [20].

For any broken URL, I attempt to find such a historical redirection to the URL's alias by inspecting all archived copies for that URL on the Wayback Machine. The challenge here is that not all redirections are necessarily correct. Some redirections correspond to soft-404s where requests for a URL redirect to an unrelated URL.

Solution: Compare redirections for similar URLs. To help differentiate valid and invalid redirections, I leverage the fact that, for any URL that it has archived, Wayback Machine almost always also has archived copies for a number of other URLs in the same directory, i.e., have the same prefix until the last '/'. To account for dates and article IDs in URLs, I ignore any numbers at the end of each URL's prefix, e.g., the broken URLs in



(a) Example good historical redirections



(b) Example erroneous historical redirections

Figure 3.4: **Comparing historical redirections of similar URLs helps me identify which redirections are soft-404s.**

Table 3.2 are considered to be in the directory "cbc.ca/news/story/".

I adapt as follows my method from §2.1.1 for detecting soft-404s. Instead of comparing the response for a target URL with that site's responses for similar randomly generated URLs, I compare the historical redirection seen in the archived copy of a target URL with temporally nearby (within 90 days on either side) 3xx status code archived copies for up to 3 other URLs in the same directory. FABLE considers the historical redirection for a URL to be correctly pointing to the URL's alias only if the target of redirection is unique compared

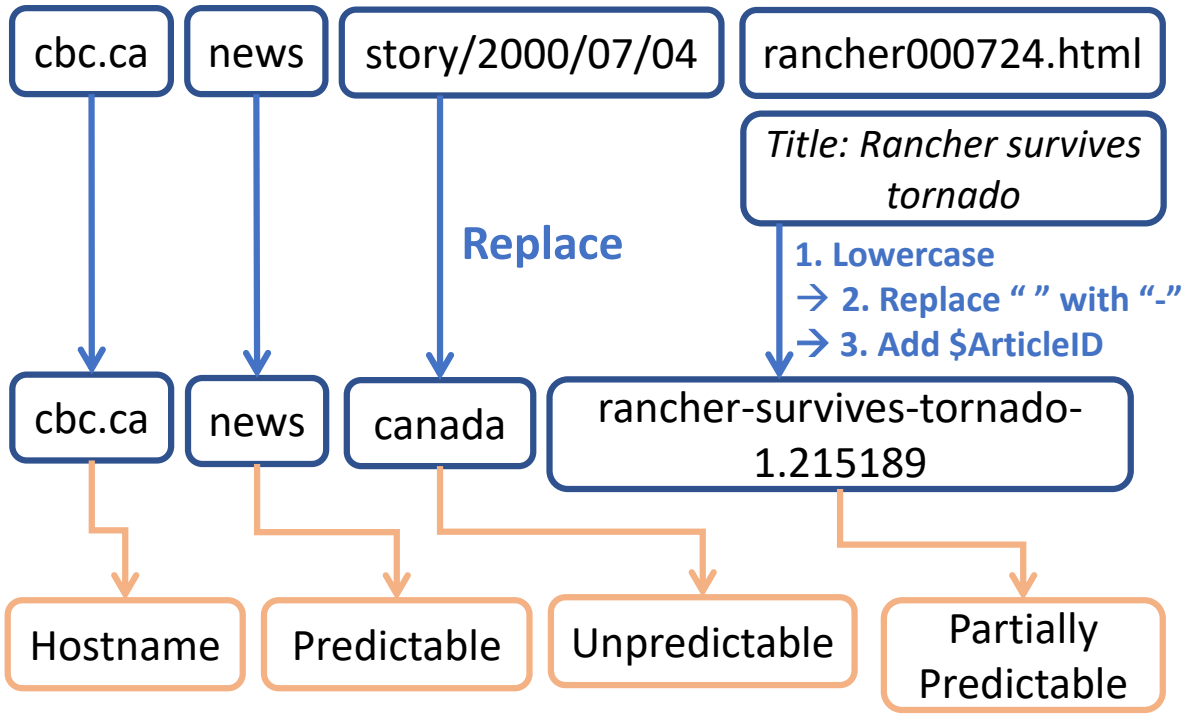


Figure 3.5: **Contrast between precise and coarse-grained characterizations of URL transformation:** How a human is likely to characterize the transformation from the third URL in Table 3.2 to its alias (blue arrows) versus the simplified pattern used by FABLE (orange arrows).

to the responses seen for other similar URLs. Figure 3.4 shows examples of correct and incorrect historical redirections identified using this method.

3.3.1.2 Match URLs to search results

When the FABLE backend does not find any correct historical redirection from a broken URL to its alias, it relies on web search using the title/content from the last archived copy for that URL. However, FABLE does not compare the archived copy's content to any live page on the same site. Consequently, for most URLs, the backend does not crawl the pages at *any* of the search results in order to determine if one of them is a match.

Approach and challenges. After grouping broken URLs that are in the same directory, for each URL u_i in a group, the backend considers the URLs $\{u_{i1}, \ldots, u_{im}\}$ of the search

URL	Alias candidate fom search results	Pattern
U1: solomon-times.com/news.aspx?nwid=1121 *Title*: "No Need for Government Candidate: CEO Transparency Solomon Islands"	*C11*: solomontimes.com/letter/1121	solomontimes.com/UP/Pr
	C12: solomontimes.com/news/no-need-for-government-candidate-ceo-transparency-solomon-islands/1121	solomontimes.com/Pr/Pr/Pr
	C13: solomontimes.com/news/governments-prime-minister-candidate-pledges-reconciliation-as-priority/1112	solomontimes.com/Pr/UP/UP
U2: solomon-times.com/news.aspx?nwid=6540 *Title*: "High Court Rules against Lusibaea"	*C21*: solomontimes.com/news/high-court-rules-against-lusibaea/6540	solomontimes.com/Pr/Pr/Pr
	C22: solomontimes.com/news/high-court-to-review-lusibaea-case/5862	solomontimes.com/Pr/UP/UP
	C23: solomontimes.com/news/lusibaea-released-opposition-uproar/5814	solomontimes.com/Pr/UP/UP

Table 3.4: **Step 1 of matching broken URLs in a directory to their aliases:** Map the URL of every search result to a coarse-grained transformation pattern. Pr: *Predictable*, UP: *Unpredictable*

results. It declares URL u_{ik} as the alias for u_i if the transformation from u_i to u_{ik} is such that other URLs in the group also have a search result with the same transformation.

To realize the above methodology, I need to characterize the transformation from every URL to each of its search results. Doing so is challenging for two reasons. First, not every portion of the new URL can be derived from the original URL and associated metadata (such as page title). For example, a new page ID may be introduced, as seen in Figure 3.5. Second, the set of transformation operations is unbounded, and the complexity of deriving the transformation between any two arbitrary URLs is exponential with respect to the number of operations in the transformation [92]. I find that even a well-optimized pattern inference engine such as Microsoft Excel's Flash Fill [44] takes over 5 seconds for the median (URL, alias candidate) pair. Therefore, deriving the transformation from every input URL to every search result for it will impose significant overhead.

Solution: Coarse-grained transformations suffice. I observe that it is unnecessary to precisely derive URL transformations because valid URLs within any site are sparsely

Pattern	URL, Candidate
solomontimes.com/Pr/Pr/Pr	(*U1*, *C12*)
	(*U2*, *C21*)
solomontimes.com/Pr/UP/UP	(*U1*, *C13*)
	(*U2*, *C22*)
	(*U2*, *C23*)
solomontimes.com/UP/Pr	(*U1*, *C11*)

Table 3.5: **Step 2 of matching broken URLs in a directory to their aliases:** (Broken URL, alias candidate) pairs from Table 3.4 with the same coarse-grained pattern are clustered together. Candidates in the top cluster are chosen by the backend as the predicted aliases.

distributed across the site's namespace. For example, given that the site cba.ca has a page whose URL ends with "rancher-survives-tornado", it is very unlikely for it to also have a page whose URL ends with "rancher_survives_tornado".

Hence, it suffices to characterize URL transformations in a coarse-grained manner as follows. When comparing two URLs – an input broken URL and the URL for an alias candidate (i.e., a search result) – I split up either URL into its components using '/' as the delimiter. I then tokenize the URL components and the page title in the input URL's last 200 status code archived copy using all non-alphanumeric characters as delimiters. I classify every component in the candidate alias URL as *Predictable*, *Unpredictable*, or *Partially predictable* depending on whether the component's tokens are a strict subset, have no overlap, or partially overlap with the set of tokens from the input URL and archived page title.[3] Figure 3.5 shows an example of how FABLE's backend maps every alias candidate URL to a sequence of predictable, unpredictable, and partially predictable components.

Within each group of similar broken URLs, once the backend establishes the predictability for each URL-candidate pair (as shown in Table 3.4), it identifies which (if any) of the candidates is the alias for each URL in the group. To do so, as shown in Table 3.5, the backend clusters candidates which are mapped to the same pattern, i.e., the same sequence

[3]To classify a URL component as partially predictable, I also require at least half of its 2-grams (i.e., every pair of consecutive tokens) to overlap. This accounts for unrelated URLs which happen to have similar tokens, e.g., "chili_peppers_camron_top_the_chart" and "red-hot-chili-peppers-attack-the-chart-116269".

Old URL	New URL	Processed URL pattern
w3schools.com/html5/ tag_i.asp	w3schools.com/tags/ tag_i.asp	w3schools.com/tags/Pr
w3schools.com/html5/ att_video_preload.asp	w3schools.com/tags/ att_video_preload.asp	
w3schools.com/html5/ html5_geolocation.asp	w3schools.com/html/ html5_geolocation.asp	w3schools.com/html/Pr
w3schools.com/html5/ html5_webstorage.asp	w3schools.com/html/ html5_webstorage.asp	

Table 3.6: **Example demonstrating challenges in using PBE.** Old URLs within the same directory on a site get mapped to multiple new directories, and not all components in the new URLs are *"Predictable"*.

of predictability for the URL's components. It picks the cluster in which the pattern has the highest number of *Predictable* and *Partially predictable* URL components. If there is a tie, the backend picks the cluster which contains candidates for more broken URLs. In rare cases, the top ranked cluster could have multiple candidates for the same broken URL. Only in such cases does the backend need to crawl the live pages corresponding to the alias candidates and check if any of them is a match based on title/content.

Of course, not all broken URLs have aliases, and the alias for a URL may not appear in the search results for it. Therefore, if the top ranked cluster 1) has only one broken URL (i.e., there is no consistent transformation pattern observed across the URLs in a group), or 2) has no *Predictable* or *Partially predictable* URL components (i.e., the alias candidates have nothing in common with the original URLs), then I declare that no aliases were found for the input broken URLs.

3.3.2 Frontend design

The most straightforward way for the FABLE frontend to leverage work done by the backend would be as follows. When, for example, a user visits a broken link, the frontend can lookup the latest archived copy of that link and issue search queries based on the title/content in that copy. The frontend can identify that search result as the alias which matches the

coarse-grained pattern associated with the top cluster identified by the backend for this URL's directory. This procedure can be completed in less than 10 seconds for the median URL (§3.4.2). It also helps preserve user privacy, since any user's frontend installation does not need to share the URLs visited by the user with FABLE's backend.

However, requiring the frontend to lookup an archived copy and issue a search query for every broken URL has several limitations. First, the frontend would be unable to find the alias for any URL with no archived copies. Second, when a user of FABLE's browser add-on visits a broken URL, if an archived copy exists and the user chooses to inspect it first, the add-on can use this period to identify the URL's alias. But, the user might choose to visit the alias without inspecting the archived copy. Therefore, the frontend must aggressively minimize the delay that it incurs in identifying the alias for any given broken URL. Lastly, even for URLs of pages that no longer exist, the frontend will unnecessarily lookup archived copies and issue web searches in the attempt to find aliases for these URLs.

To deal with these shortcomings, in addition to the coarse-grained transformation pattern for every directory, the FABLE backend shares two additional takeaways from its work in finding aliases.

3.3.2.1 Transformation programs to infer aliases

Approach and challenges. First, given a broken URL, the frontend attempts to locally generate the alias without performing any web search. To enable this, whenever feasible, the backend uses the aliases that it discovers for the URLs in a directory to learn a precise URL transformation function for that directory.

I find programming by example (PBE) [88] well-suited for this purpose. The PBE approach entails providing a number of input-to-output examples, using which a program can be synthesized to transform input to output. Figure 3.6 shows an example of how a program generated from several (URL, alias) pairs can be used by the frontend to then locally predict the alias for any other broken URL in the same directory. I leverage Microsoft's PROSE

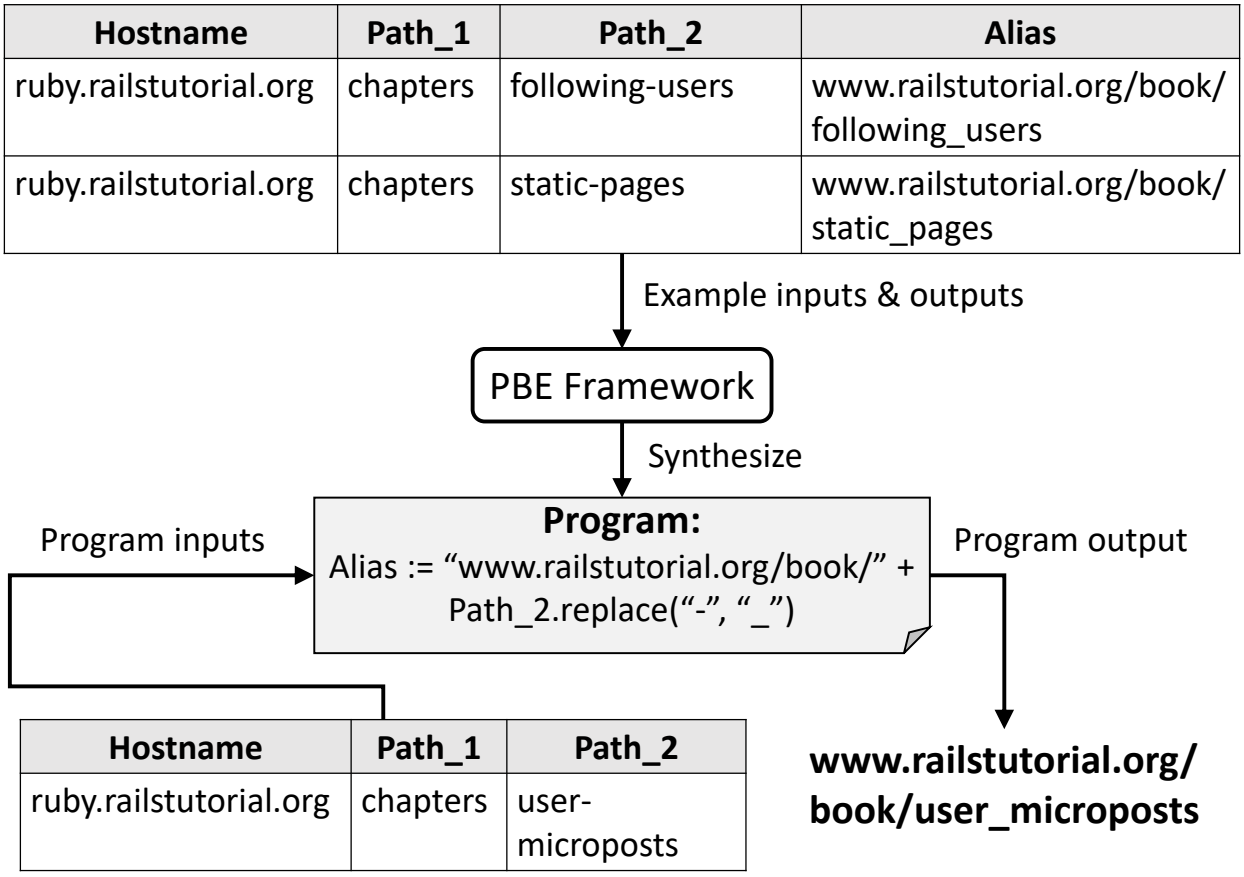


Figure 3.6: **Programming by example in action.** PBE framework takes in example inputs and outputs to learn a program, which can then be used to similar inputs into output.

PBE framework [39], which is the prediction engine underlying Excel's Flash Fill [44]. For every broken URL, apart from tokens in the URL, the program also takes as input auxiliary information such as page title and date of creation [37], which I extract from archived copies of that URL, when available.

The question then is: when and how to use PBE to enable the FABLE frontend to locally generate the alias for any given broken URL? On the one hand, it is not always feasible to use this approach, e.g., in cases like in Figure 3.5 wherein a page's new URL includes a new page ID. On the other hand, attempting to learn a URL transformation program only when all components are deemed *"Predictable"* in the coarse-grained pattern for a directory would be too restrictive. For example, for the broken URLs in Table 3.6, it is intuitive to see the

feasibility of learning the underlying transformation pattern, but some of the tokens in the new URLs (e.g., *"tags"*, *"html"*) are not derived from the original URLs. Moreover, PBE relies on learning a single program based on all inputs. Whereas, as seen in Table 3.6, URLs in the same directory can end up getting mapped to new URLs in different directories.

Solution: Group example inputs with similar aliases, and learn one program per group. I address all of the above-mentioned challenges as follows. First, I observe that learning a single program for all broken URLs in a directory suffices when all URL components are *"Predictable"* in the coarse-grained pattern learned for this directory. When there are *"Partially predictable"* or *"Unpredictable"* components, there is ambiguity about whether multiple programs need to be inferred or no precise transformation function can be learned. To distinguish between these cases, second, I split up the broken URLs in a directory such that all aliases in a partition have the same prefix. If the URL components following the common prefix are not predictable, like for cbc.ca's URLs in Table 3.2, then we declare that no precise URL transformation program can be learned. If the components other than the prefix are predictable, like in Table 3.6, then I attempt to infer multiple programs for the same directory, one for each partition. Given a broken URL, the frontend applies every program learned for this URL's directory, and at most one of these inferred aliases will prove to be a functional URL.

3.3.2.2 Identifying deleted pages

The second optimization in FABLE's frontend is to skip doing work for URLs which are broken because the pages they previously led to no longer exist.

For this, I observe that, even when FABLE's backend is unsuccessful in finding aliases for the first few URLs within a directory, this provides valuable information. Specifically, for the first four URLs in a directory, if FABLE neither 1) finds any alias, nor 2) finds any candidate where the last URL component is either *Predictable* or *Partially predictable*, then I empirically observe that it finds no aliases for any other URLs in that same directory with

high likelihood. The backend shares with every frontend all such directories in which it believes attempting to find an alias for the URLs in those directories is futile.

Thus, when a FABLE frontend encounters a broken link on the web, it first checks if the URL matches a directory that likely corresponds to a set of deleted pages. If not, it applies any URL transformation programs applicable to this directory and checks to see if any of them yields a URL corresponding to a live page. If still unsuccessful, the frontend eventually falls back to issue a search query and checks if any result matches the coarse-grained pattern for aliases identified for this URL's directory by the backend.

3.4 Evaluation

I evaluate FABLE from three perspectives: 1) its ability to correctly discover the new URLs for broken links which do have functional aliases, 2) its efficiency in finding aliases, and 3) the utility of directing users to the aliases it finds instead of serving archived page copies. I compare FABLE with two classes of prior solutions, described earlier in §3.1: 1) content hash-based addressing to make URLs resilient, and 2) reliance on similarity in page content/title to rediscover missing pages. I refer to my implementations of these two classes of solutions as ContentHash and SimilarCT.

The main takeaways from my evaluation are:

- When I compare FABLE to prior solutions on a dataset of 20K broken external links sampled from pages on Wikipedia, Medium, and Stack Overflow, FABLE finds correct aliases for 51% more broken links.
- Compared to SimilarCT, FABLE's backend needs to crawl only $\frac{1}{20}^{th}$ as many pages from the live web and issue $\frac{2}{3}^{rd}$ as many search engine queries to find aliases for the same number of broken URLs.
- For the median broken URL, FABLE's frontend completes its attempt to find the URL's alias in less than 10 seconds. This is more than 30 seconds lower than the latency with

SimilarCT.

- My analysis of a random sample of 100 broken URLs for which FABLE finds aliases highlights the importance of not always relying on archived page copies: 9% have not been archived by Wayback Machine, 24% have stale content, and 70% include functionality that does not work on archived copies.

3.4.1 Coverage and Accuracy

I begin my evaluation by considering two questions: How many broken URLs is FABLE able to find aliases for? What fraction of the aliases it finds are correct? Given the lack of ground truth data mapping old page URLs to the corresponding new URLs, prior work on rediscovering web pages [89, 97] only considered: given a cached copy of a live web page, do the search queries that they extract from that copy yield that page in the top few search results? To instead perform a more realistic evaluation, I first assemble a dataset of 1000 broken URLs where I know the correct alias (if one exists) for every URL. I then apply FABLE's techniques on a larger dataset of 20K broken URLs where I lack ground truth.

3.4.1.1 Ground truth dataset

My ground truth dataset comprises two sets of 500 dysfunctional URLs each, which exploit site operators' intent evident in how those sites either previously responded or currently respond to requests for these URLs.

- I compile the first set (*Alias*) using a manually verified set of historical redirections, i.e., for every broken URL in this set, via manual examination of its archived copies on the Wayback Machine, I can confirm that the URL that it previously redirected to is indeed its alias.
- The second set (*NoAlias*) comprises URLs that previously pointed to pages which likely no longer exist now. I identify such URLs as ones that today elicit 410 status code responses, which indicate that *"access to the target resource is no longer available at the origin server*

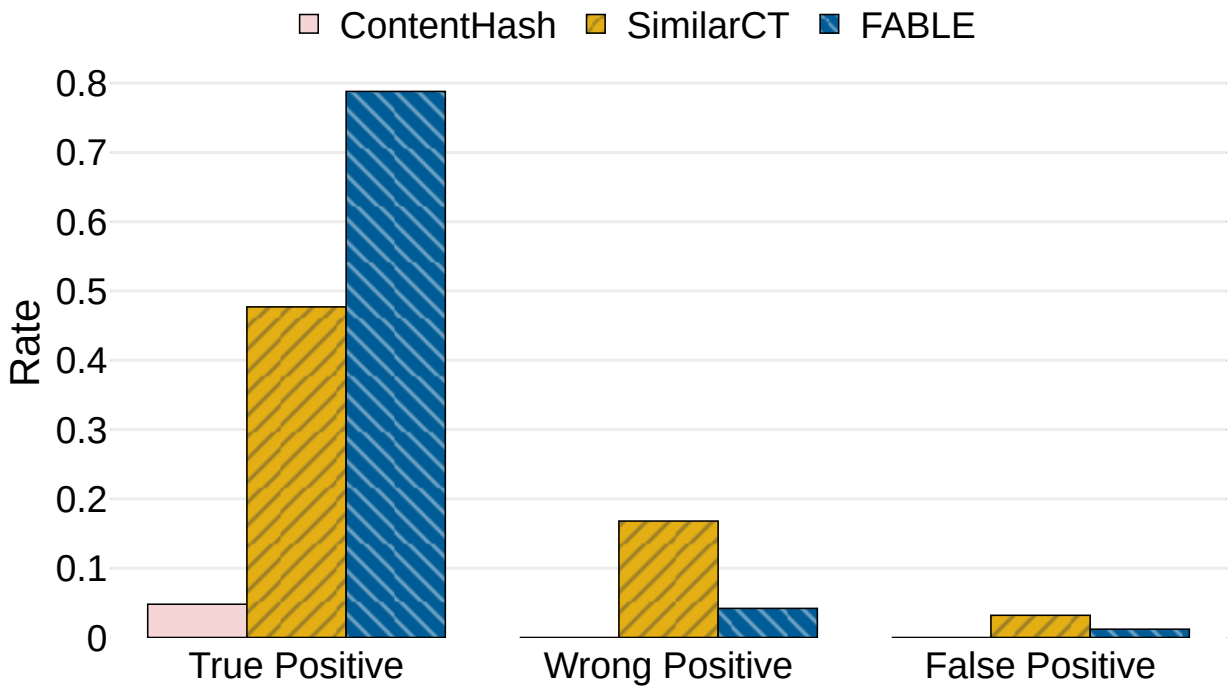


Figure 3.7: **Compared to alternate approaches,** FABLE **significantly improves coverage and accuracy.** On a dataset of 500 URLs for which aliases are known and 500 URLs which likely have no alias, the fraction of true positive, wrong positive, and false positive matches found by FABLE and existing approaches.

and that this condition is likely to be permanent" [26]. While I cannot confirm whether or not any particular 410 response is erroneous, whenever I do find an alias for any of these URLs, I manually check its correctness.

Since my knowledge of aliases here is based on redirections found on Wayback Machine, I withhold 3xx status code archived copies from FABLE when running it to find aliases for broken URLs in this dataset. For the prior approaches that I compare against, I optimize their performance as follows.

- For ContentHash, I filter out every page's boilerplate content using Chrome's distiller [31] before computing hashes. This helps eliminate any changes caused by page templates, content recommendations, etc.
- For SimilarCT, as I mentioned previously, prior work leaves it to the user to decide if any of the search results for a URL is that URL's alias. To instead offer the same interface as

FABLE, I consider a search result to be a match if it is the only one whose title or content has a TF-IDF similarity [127] of 0.8 or more with the archived copy; prior work [119] recommends that a threshold of 0.8 is appropriate to detect if a page has been only slightly modified.

Figure 3.7 shows the true-positive rate (fraction of *Alias* URLs which are matched to their known alias), wrong-positive rate (fraction of *Alias* URLs matched to an incorrect alias), and false-positive rate (fraction of *NoAlias* URLs for which a match is found).

True positives. First, FABLE achieves a higher true-positive rate than prior approaches – close to 80% versus less than 50% – because of 1) its reliance on URL transformations, not content similarity, to match broken URLs to search results, and 2) its inference of aliases for URLs which have either never been archived or whose alias does not appear in the search results. For example, both existing approaches fail to match the broken URL https://www.udacity.com/courses/cs262 to its alias http://udacity.com/course/programming-languages--cs262; since the class has been updated over time, both the title and content of the page have significantly changed. In contrast, FABLE is successful because it observes search results with the same URL transformation for other classes with similar URLs, e.g., https://www.udacity.com/course/2d-game-development-with-libgdx--ud405 is one of the alias candidates for http://udacity.com/courses/ud405. Moreover, FABLE successfully discovers the vast majority of true positives found by SimilarCT; for only 4.4% of the 500 URLs in the *Alias* dataset, SimilarCT matches a broken URL to its correct alias without FABLE being able to do the same.

Note that the true-positive rate that I observe for SimilarCT is significantly lower than that claimed by prior work [97]. I believe this is because prior work evaluated their techniques on functional URLs and "pretend" that they are broken. In contrast, my ground truth dataset consists of URLs that are truly broken. Since broken URLs are likely to have fewer recent archived copies than functional URLs, there are potentially many more changes between the last archived copy and the live page.

I find that the following factors prevent FABLE from achieving a true-positive rate of 100% on this dataset.

- First, FABLE fails to match 3% of the *Alias* URLs to their known alias due to the incompleteness of Google's and Bing's search indices; the aliases for these URLs do not appear in the search results even though the archived copy from which FABLE formulates search queries closely matches the live page at the alias URL today, i.e., TF-IDF similarity between the latest archived copy and the live page is more than 0.8.
- For another 8% of the *Alias* URLs, neither does the alias appear in the results for FABLE's search queries (because content on the live page has been updated significantly since when it was last archived) nor is FABLE able to infer the alias (because it could not learn any precise URL transformation program from its attempts to find aliases for other URLs in the same directory).
- The remaining missed aliases can be attributed to FABLE's procedure for matching every URL to one of the search results for it. Either FABLE's criteria end up being too strict (for 6% of *Alias* URLs), or it picks incorrectly between multiple observed URL transformation patterns and matches the URL to an incorrect alias (for 4% of URLs).

Wrong and false positives. Second, SimilarCT results in more wrong-positives than FABLE because the former can end up matching a broken URL to a different page that happens to have similar content, like in the example from marvel.com that we discussed in §3.1. FABLE avoids doing so by examining other broken URLs in the same directory and their candidates, e.g., FABLE correctly chooses https://www.marvel.com/comics/issue/22962/what_if_2008_1 as the alias. Although ContentHash has no wrong-positives or false-positives, its low true-positive rate means it has little practical value for making links to web pages resilient.

Finally, FABLE's false-positive rate is quite low at ~1%. These false matches occur when, for a bunch of similar URLs, wrong candidates which share tokens with the corresponding broken URLs appear in the search results. For example, FABLE incorrectly declares https:

//exclaim.ca/music/article/black_mountain-wilderness_heart (an article introducing a band's new album) as the alias for http://exclaim.ca/Contests/black_mountain_wilderness_heart/ (a music contest regarding the same band) because a similar broken URL, http://exclaim.ca/Contests/we_are_city-violent, has a search result with the same change: https://exclaim.ca/music/article/we_are_city-violent. Most of the false positives identified by FABLE are also flagged by SimilarCT; only 0.4% of all *NoAlias* URLs are uniquely matched by FABLE to an incorrect alias.

3.4.1.2 Broken links from three sites

Next, I study FABLE's coverage and accuracy on a larger dataset of broken URLs for which I lack ground truth information. I 1) consider all the broken external links I previously observed on Wikipedia, Medium, and Stack Overflow (Table 2.1), 2) group them into directories, 3) choose all URLs in directories with ≥ 5 URLs (as seen in Figure 3.1, for 80% of broken URLs, at least 4 other URLs in the same directory also stopped working when they became dysfunctional), and 4) pick all URLs that point to functional sites with English content. I focus on links to English sites because my current implementation can compute TF-IDF similarity only for English text. Though FABLE seldom relies on comparing the title/content of pages, SimilarCT is crucially dependent on this capability. Given an implementation for calculating the similarity between text in other languages, I can easily extend my implementations of FABLE and SimilarCT to those languages.

From this dataset, I randomly sample 20,000 broken URLs, comprising 12,000, 4,200, and 3,800 broken external links from pages on Wikipedia, Medium, and Stack Overflow, respectively. Given the low coverage seen on the ground truth dataset with ContentHash, on this dataset, I focus on comparing FABLE with SimilarCT.

Precision. The ideal way to address the lack of ground truth information in this dataset would be to contact the operators of the sites which host the broken URLs I consider. However, such an approach is impractical given that the URLs in my dataset are spread

across a large number of sites and the operators of most sites are unlikely to respond to my queries.

Therefore, I instead evaluate the quality of aliases from users' perspective, i.e., do users consider the live page that currently exists at the FABLE-identified alias for a broken link to serve the purpose that the page previously available at that link was meant to serve? The alias for a broken link is useful to users when either archived copies are not an adequate replacement for the live page, or no archived copy exists for that link. Since I lack an automated approach to identify the former at scale, I focus on the latter.

I asked members of the Wikipedia community to check a random sample of the aliases found by FABLE for broken references on Wikipedia which have been marked as "permanently dead" [30]. For these links, not only is the linked page no longer accessible via the original link, but no archived copy exists for users to refer to. I randomly selected 103 aliases found by FABLE for permanently dead links (selecting at most 2 per domain) and posted them on a public Wikipedia page [43]. To aid users in the task of verifying the aliases, my page also listed auxiliary information for every link such as the Wikipedia article in which the link appears and the anchor text of the link. I then solicited the Wikipedia community to check these aliases and share the observations they made in doing so.

Wikipedia users determined that 89 of the 103 aliases are correct, 6 are incorrect, and they were unsure about the remaining 8. An example URL for which they were uncertain about the correctness of the alias is http://igokisen.web.fc2.com/kl.html, which a Wikipedia user linked to in 2011 to point others to results from that year's edition of the Korean Baduk League. That URL no longer works today, and FABLE identified http://igokisen.web.fc2.com/kr/kl.html as the alias. But, the page at the alias lists the results from the most recent edition of the league. Since there is no archived copy for the original link, it is hard to tell whether the user who originally created the reference did not link to an appropriate page, the content on the page has drifted, or the alias found by FABLE is incorrect. Depending on whether I pessimistically consider all the ones marked unsure as incorrect or optimistically

Source	DNS+	404	Soft-404	**Total**
Wikipedia	1414	7458	3128	**12000**
Medium	737	2127	1336	**4200**
Stack Overflow	413	2270	1117	**3800**
% Alias found	15.8%	23.0%	27.9%	23.4%

Table 3.7: **In my selected sample of broken external links on Wikipedia, Medium, and Stack Overflow,** FABLE**'s success rate in finding the alias as a function of how the URL is broken.** DNS+ means failed DNS resolution or TCP/TLS connection setup, and Soft-404 includes URLs which redirect to an unrelated page.

	≤'10	'10–'15	'15–'21	**Total**
No. URLs	1846	5754	6414	**14014**
% Alias found	25.1%	31.5%	31.5%	30.6%

Table 3.8: FABLE**'s success rate in finding the alias for a URL as a function of how long it has been since that URL stopped working.** For URLs archived by Wayback Machine before they became dysfunctional, year of last archived copy with a 200 status code and fraction for which FABLE found aliases.

consider them as correct, FABLE's accuracy is between 86% and 94%, i.e., 90% on average.

Coverage. FABLE finds aliases for 4680 of the 20K URLs in my dataset. Table 3.7 shows that the fraction of broken links for which FABLE discovers aliases varies from 18.8% on Medium to 26.2% on Stack Overflow. My success in finding aliases for more of the URLs found on Stack Overflow stems from two factors. First, as seen earlier in Figure 2.1, the URLs found on Stack Overflow are biased toward higher ranked domains, most of which are categorized as 'Computers & Electronics'. I hypothesize that URLs on these domains are more likely to break due to site reorganizations, not page deletions. Second, my ability to find aliases for broken URLs found on Medium is hampered by the fact that a much lower fraction of these URLs have archived copies, in contrast to URLs found on Wikipedia and Stack Overflow.

Table 3.7 also shows that FABLE's success rate is lower for URLs which fail to resolve as compared to URLs which result in erroneous redirections. Moreover, FABLE's ability to find aliases is not limited to URLs that stopped working recently; Table 3.8 shows that when I

compare URLs which were last successfully archived 7–12 years ago to those that became dysfunctional more recently, FABLE's success rate is comparable.

SimilarCT finds aliases for 4286 URLs, close to the number of aliases found by FABLE. However, what matters is the number of *correct* aliases found. Taking into account FABLE's precision of 90% and its true-positive rate of 79% on the ground truth dataset, I can estimate the total number of broken URLs in this dataset that do have an alias. Using SimilarCT's metrics from the ground truth dataset, I can then estimate that the number of correct aliases discovered via this approach is only 2786. Thus, compared to SimilarCT, FABLE correctly unearths aliases for 51% more broken URLs.

Table 3.9 shows the primary factors limiting FABLE's ability to find aliases for more broken links. First, the dominant bottleneck for search-based alias discovery is that none of the search results are considered as a match. This might occur because either the page is absent from the search indices of both Google and Bing, FABLE's criteria for considering a search result as a match is too strict, or the page simply no longer exists. I am unable to distinguish between these factors because I do not know which of the URLs in this dataset do have functional aliases. Second, the absence of 3xx status code archived copies and copies with erroneous redirections are both equally responsible for limiting the utility of historical redirections. Lastly, when FABLE fails to infer the alias for an URL, this is either because it did not find any aliases for other similar URLs (suggesting that all of these pages no longer exist) or because the new URLs contain either partially predictable or unpredictable tokens (which prevents me from programmatically inferring them).

3.4.2 Efficiency

Next, I evaluate FABLE's efficiency from two perspectives: the backend's throughput, and latency at the frontend. To ensure a fair comparison, I focus only on URLs for which SimilarCT has a chance of finding the alias. These are URLs for which 1) 200 status code archived copies exist, and 2) the alias identified by FABLE appears in the search results. I

Search	No. URLs
No valid archived copy	5629
No search results	1541
No matching search result	8195

Historical redirection	No. URLs
No 3xx archived copy	7890
Erroneous 3xx archived copy	7475

Inference	No. URLs
Not enough examples to infer	12650
Pattern not possible to learn	2790
No good alias inferred	15

Table 3.9: **Breakdown of reasons for FABLE's inability to find aliases using different methods.**

then assume that SimilarCT matches each such broken URL to the same alias as the one found by FABLE.

Backend throughput. As shown in Figure 3.8, compared to SimilarCT, FABLE dramatically reduces the number of page crawls from the live web, by as much as 23x. Recall that FABLE's backend needs to crawl the page at any of the search results only in the rare case when multiple URL transformation patterns appear credible. FABLE also reduces the number of search engine queries by a third.

Frontend latency. Figure 3.9 shows that SimilarCT takes close to 40 seconds for the median URL. This is because of having to crawl pages linked from the search results one at a time until a match is potentially found. Since all the search results for a URL are from the same site, crawling pages at all of the top 10 results in parallel is not an option; doing so would typically exceed the rate at which a website permits its pages to be crawled.

In contrast, FABLE's latency for the median URL is significantly lower, irrespective of whether it finds an alias for that URL or the method it uses to find the alias. When FABLE finds an alias via inference, the median runtime is less than 5 seconds. Even when FABLE fails to find any alias for a URL, it takes half as long as SimilarCT due to my policy of bailing

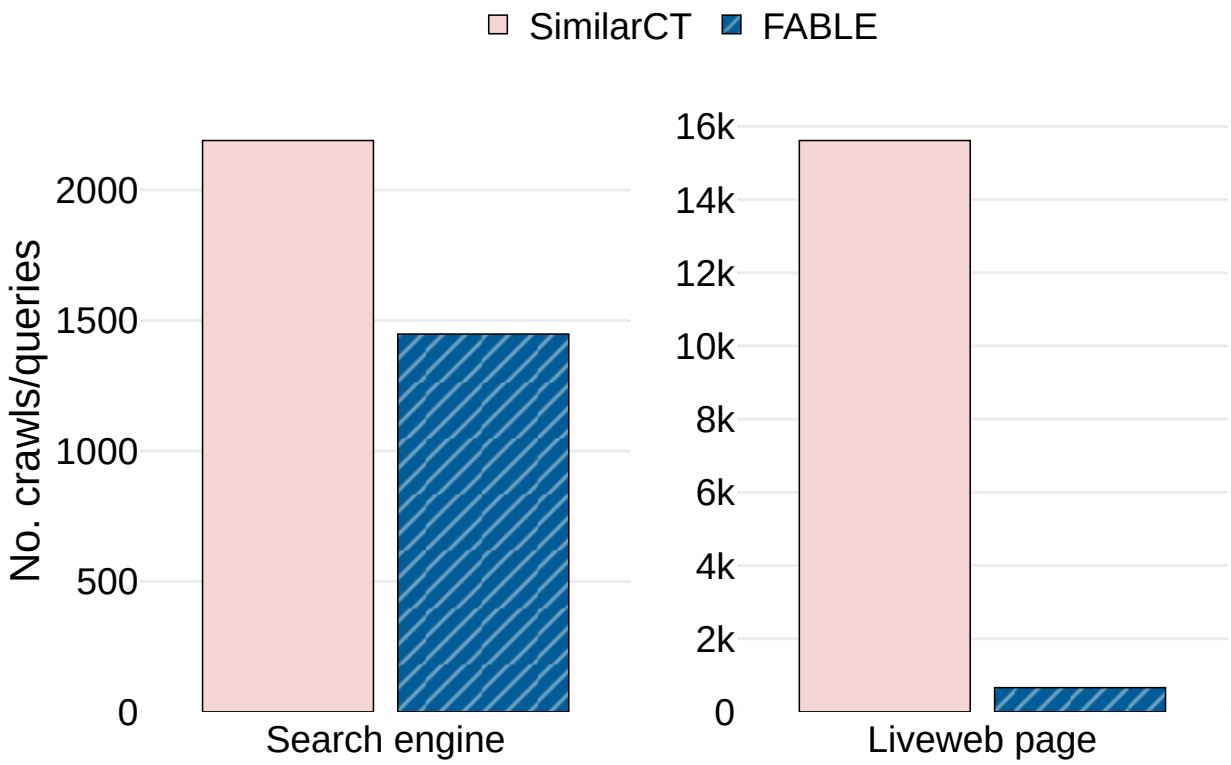


Figure 3.8: **Efficiency of** FABLE **backend.** For 1000 broken URLs, in comparison to SimilarCT, FABLE significantly reduces both the number of web search queries issued and pages crawled from the live web.

on URLs in a directory once I determine that none of the URLs in that directory are likely to have aliases. In fact, FABLE typically finds the alias with less delay than that involved in loading an archived copy from the Wayback Machine. As a result, in the common case, FABLE's frontend would have identified a broken link's alias (if it exists) before the user finishes examining the archived copy of that link.

Systems that support content-based lookups offer lower latency than FABLE, e.g., IPFS [132] reports median latencies of less than 3 seconds. But, the low coverage with this approach undermines its utility.

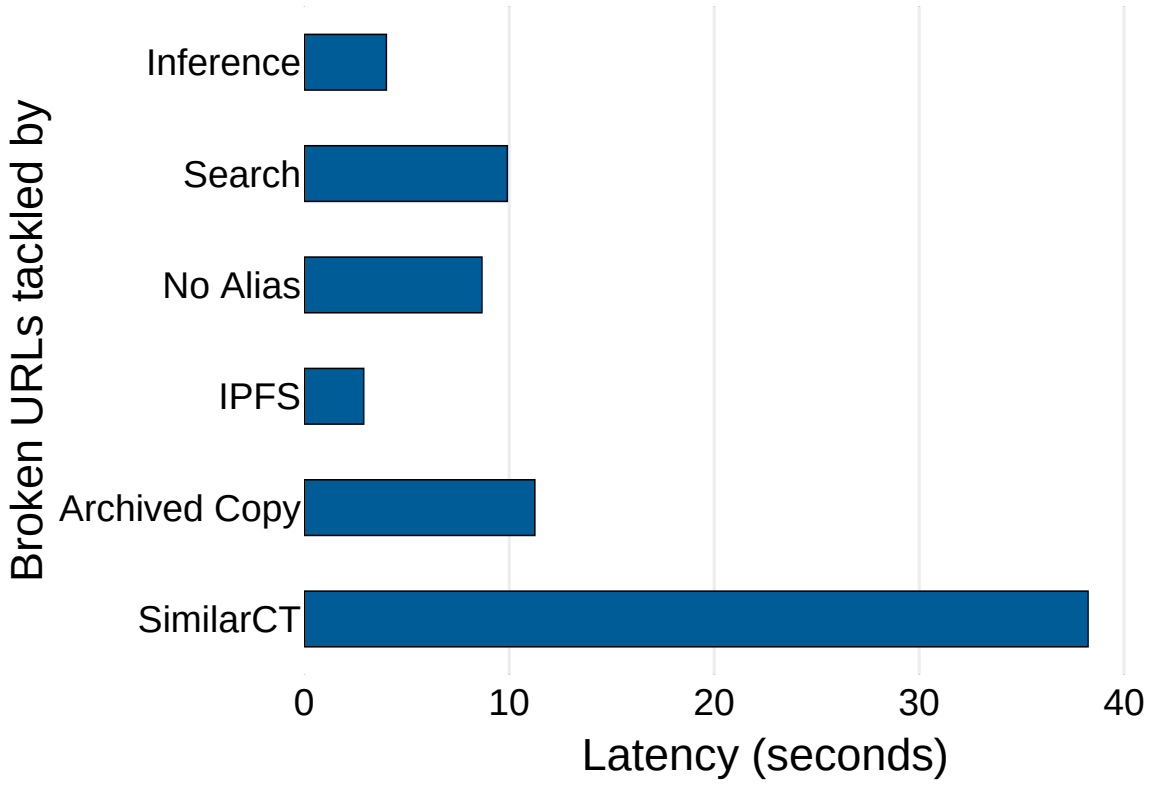


Figure 3.9: **Median latency incurred at the FABLE frontend as a function of whether or not it finds the alias and the method it uses when successful.** I compare to the median latency for loading a page from either IPFS or the Wayback Machine, or for finding a broken link's alias using SimilarCT.

3.4.3 Utility of Aliases

In the last part of my evaluation, I demonstrate the utility of aliases discovered by FABLE, as compared to relying on archived page copies.

When a user visits a broken URL, if that page is still available at an alternative URL, directing the user to the new URL is beneficial in many ways as compared to serving an archived copy of the broken URL. To study how commonly these benefits apply, I pick 100 random URLs for which FABLE found correct aliases. For each URL, I manually compare Wayback Machine's last successful archived copy for that URL to the page found today at the alias URL. Table 3.10 shows that, for almost all of these 100 broken URLs, reliance on archived copies would be suboptimal for users. Users will benefit from the aliases identified by FABLE mainly in two ways.

		# URLs
Downsides for users	No archived copy	9
	Stale content	24
	Service not usable	70
Downsides for site providers	Loss of recommendations	60
	Loss of ad revenue	45
	Total	**93**

Table 3.10: **Across 100 broken URLs for which FABLE finds an alias, breakdown of reasons why reliance on web archives to serve requests for these URLs would be undesirable.**

- It is common for pages to offer functionality such as login, subscription, comments and feedback. For example, https://www.php.net/manual/en/function.setlocale.php, which is the alias of http://de3.php.net/manual/en/function.setlocale.php, enables users to leave notes. Such functionality cannot work on any archived copy.
- On many pages, since the main content (i.e., content left after ignoring boilerplate [101]) is updated over time, the last successful archived copy contains stale content. For example, the broken URL http://www.athleticsweekly.com/stats/records/world-records-and-best-performances-womens-indoor/ was pointing to a page that lists current world records in women's track and field. As time goes on, the records will change, as reflected at the alias https://athleticsweekly.com/stats/records/world-records-best-performances-womens-indoor/.

When a page still exists on the web, relying on an archived copy of the page is undesirable not only for users, but also for web providers. For example, when a link to a page is broken, the provider of that page might lose out on ad revenue and the ability to attract the user to their site by recommending other relevant pages. Table 3.10 shows that these downsides too commonly apply.

3.5 Summary

Given the prevalence of dynamic content and app-like functionality on the web, my work calls for a rethink of the common practice of coping with any dead link by using an archived copy. I demonstrated that a sizeable fraction of broken URLs on the web are a consequence of site reorganizations, and that it is indeed practical to automate the discovery of the new URLs for these pages. I believe the aliases found by FABLE can help significantly improve the long-term availability of information and services on the web.

CHAPTER 4

Toward Better Efficiency vs. Fidelity Tradeoffs in Web Archives[1]

In the previous chapter, I addressed link rot by reviving dead links through their aliases. However, not all broken links have aliases–the associated pages may have just been deleted. In such cases, users can only rely on archived copies to access the missing content. It is therefore important to ensure that these archived copies are delivered with high fidelity.

In this chapter, I target the challenge web archives face in efficiently archiving modern web pages while preserving their fidelity. Since today's web increasingly relies on JavaScript to progressively build content during page load, preserving fidelity often requires more than traditional static crawlers; it demands browser-based crawlers to ensure all necessary resources are fetched. However, the high computational overhead of browser-based crawlers makes it impractical for archives to use them on every page.

To help web archives operators come up with better tradeoffs between archived copies fidelity and archiving efficiency, I make the case that a web archive does not have to make a binary choice between dynamic or static crawling. Instead, by using a browser for a carefully chosen small subset of crawls, an archive can significantly improve its ability to serve statically crawled pages with high fidelity. First, I show how to reuse crawled resources, both across pages and across multiple crawls of the same page over time. Second, by leveraging a

[1]The content of this chapter is revised from the previously published publication "Toward Better Efficiency vs. Fidelity Tradeoffs in Web Archives [143]"

dynamic crawl of a page, I show that subsequent static crawls of the page can be augmented to fetch resources without executing the scripts which request them. I estimate that, as long as 8.9% of page crawls use a browser, an archive can serve roughly 99% of the remaining statically crawled pages without any loss in fidelity, up from 55% without our techniques.

4.1 Dataset and Approach

A web archive crawls pages from the live web, and records all crawled resources in Web ARChive (WARC) files [15]. Over time, it may crawl the same resource multiple times. Therefore, the archive indexes each copy of a resource by the combination of the resource's URL and the timestamp at which this copy was crawled.

When a user wishes to load a snapshot of a desired page, of all the times at which the archive crawled this page, the user can pick one. For example, a user can load an archived page from the Wayback Machine by specifying a URL of the form https://web.archive.org/{collection}/{timestamp}/{page-URL}. For every resource that the user's browser requests as part of executing this page load, the archive looks up its index to return that copy whose timestamp is closest to the one specified in the request.

4.1.1 Data collection

To quantify the deficiencies of statically crawling pages and to understand how other dynamic crawls can help address those deficiencies, I collected two datasets. Both datasets contain pages from sites which span a broad range of site rankings. Whether my findings from these datasets are representative of the web at large is open to further study.

Longitudinal dataset. Starting in January 2025, I crawled 10K pages each week for 16 consecutive weeks. I selected 20K sites based on their ranking in the Tranco [122] dataset: 2K, 3K, 5K, and 10K sites, respectively, from the rank ranges [1, 10K], [10K, 100K], [100K, 500K], and [500K, 1M]. For each site, I selected the homepage along with four other internal pages sampled at random from all the live pages on this site for which archived copies exist

on the Wayback Machine. This yielded a total pool of 100K candidate pages, from which I randomly selected 10K pages.

Cross-sectional dataset. I also collected a dataset spanning fewer sites, but with more pages per site. From the original pool of 20K sites, I selected 1,000 sites at random. On each site, I crawled 50 live pages, again sampled at random from the Wayback Machine. I crawled these pages once on the 3rd and once on the 4th of May 2025.

In both datasets, I crawl each page using Chrome with Webrecorder's extension [62] enabled, which saves all the crawled resources in a WARC file. When loading a page, my crawler also mimics user interactions (e.g., mouse click and hover), as some of these can trigger additional resource fetches. I 1) collect all event listeners attached to visible elements, 2) group event listeners by their handler functions and the classes of their associated elements, and 3) select one interaction from each group and trigger the corresponding event handler.

To compare static and dynamic crawls of pages, I would ideally recrawl every page using a static crawler. However, even back-to-back loads of a page often differ significantly in the set of resources. Therefore, for every page snapshot that I crawl using Chrome, I construct a corresponding static crawl comprising all the resources fetched by Chrome except those fetched via JavaScript.

I use pywb [17], a widely used open-source archival replay system, for loading my crawled page snapshots. I configure the browser to trust a proxy run by pywb [60], which ensures that the browser's request for any resource is served from my local copy, rather than from the original server which hosted it.

4.1.2 Fidelity assessment

For every static crawl of a page, I seek to compare it to the corresponding dynamic crawl and determine whether loads of the two copies differ. Simply checking if loading the statically crawled copy results in more failed resource fetches is not enough. The resources that a browser is unable to fetch may correspond to functionality which will not work on an archived

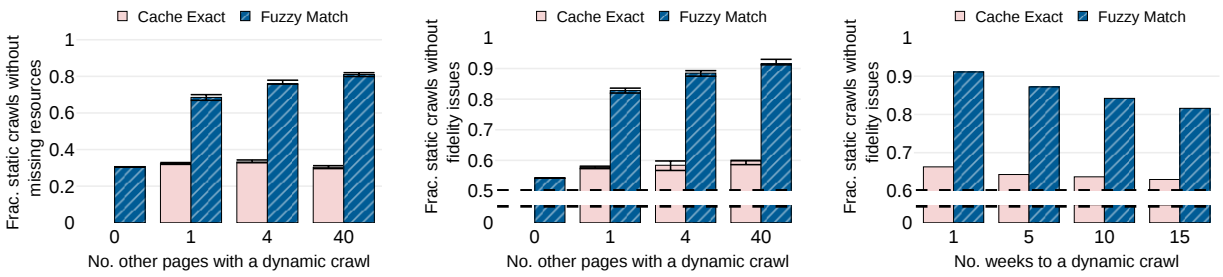


Figure 4.1: **As a function of the number of other pages for which dynamic crawls from at most a day apart are available, fraction of static crawls which** (a) have no missing resources, and (b) do not violate fidelity. Error bars show variance across 3 trials. (c) Fraction of static crawls which do not violate fidelity when dynamic crawls of the same page from another time are available.

copy of a page (e.g., analytics or push notifications).

Instead, existing archival systems typically rely on a comparison of screenshots for quality assurance [58]. They take a screen capture of a live page when crawling it, and do the same when loading the archived copy. If the screenshots match, then we can be assured that any failed resource fetches in the latter load are inconsequential.

But, due to the preponderance of dynamic content on modern web pages, I find that screenshots often differ even when the archived copy perfectly preserves fidelity. For example, on the homepage of news websites such as cnn.com, a video which shows the latest headlines automatically begins playing. Depending on the timing of the screen capture when an archive crawls the page versus when the corresponding crawled copy of the page is loaded, the screenshots can differ despite no loss in fidelity.

To ensure that dynamic page components do not affect my assessment of the fidelity of archived pages, I instead use the browser's representation of a page's visual appearance. When a modern browser renders a page [50], it fetches the page's resources, and constructs the DOM and CSSOM trees from HTML, CSS, and executed JavaScript. It then compiles DOM and CSSOM into a *layout (or render) tree* which specifies the dimensions and positions of all visible elements on the page. On any page which contain GIFs, animations, and videos, the pixels painted on the screen continuously change. However, the corresponding elements

in the layout tree remain stable.

Therefore, I compare the dynamic and static crawls of a page by examining the layout tree in the two loads. The layout tree is an internal browser data structure [59] which cannot be accessed directly. To approximate it, once the browser's network is idle for 5 seconds, I 1) traverse the DOM tree to identify visible elements, and 2) record each element's computed style, including dimensions, position, and attributes that affect composition and painting. I consider two copies of a page as equivalent if their layout trees match after the initial page load as well as after each interaction.

I use the pages from my longitudinal dataset to evaluate the use of layout tree comparison for detecting fidelity violations. For each page, I extract layout trees from two page loads in Chrome: when loading the page from the live web, and when replaying the corresponding static crawl using pywb. I sample 100 pages where my comparison reports a mismatch, and 100 where it reports no differences. Two of the checkers independently checked each page. Each checker confirmed if the cases in which differences had been identified were indeed valid. When the two loads were reported to be a match, each checker spent at least a minute searching for potential differences.

My inspection revealed that the comparison of layout trees results in a precision of at least 91% (i.e., 91 of the 100 reported mismatches correspond to actual fidelity violations) and recall of at least 96% (i.e., 96 out of 100 reported matches indeed have no issues). False positives arise because, on pages with JavaScript-controlled dynamic components, the layout tree will differ depending on when it is captured, e.g., the source image in the DOM element corresponding to a carousel will vary over time. On the other hand, because the layout tree only captures the properties of visible DOM elements match, it fails to account for unfetched images.

In comparison, though screenshot comparison has 100% recall, it identifies fidelity issues with a precision of only 24%, i.e., 76% of the pages where the screenshots of the live and archived page differ correspond to false alarms. Moreover, compared to screenshots,

collecting layout trees incurs lower crawling overhead. Relative to a dynamic crawler which captures neither, I find that crawlers which collect screenshots or layout trees reduce crawling throughput by 13% and 4%, respectively.

My fidelity assessment for a page load from a static crawl relies on having its dynamic crawl counterpart as a reference for comparison. In real-world web archives, however, such a ground truth is typically unavailable, making it difficult to evaluate whether a statically crawled archived copy truly preserves the fidelity of its corresponding live page. While determining this with absolute certainty is challenging, there exists a criterion that, when satisfied, guarantees fidelity preservation (though the converse does not hold). Specifically, if during the loading of an archived copy, all requested resources are both available (i.e., respond with HTTP 200) and up to date–indicated by headers such as `ETag`, `Last-Modified`, `Cache-Control`, and `Expires`, whose validity periods include the crawl timestamp–then the archived copy can be considered to have preserved the fidelity of the original page. The criterion can be particularly valuable in cases where demonstrating that an archived copy faithfully reproduces the original page is critical, such as for legal and evidentiary use.

4.1.3 Static vs. Dynamic

To compare the loads of statically crawled and dynamically crawled page snapshots, I use the first week of my longitudinal dataset. For 54.2% of the 10K pages in my datasets, loads of the copy crawled without a browser are the same as that crawled with a browser. I find that such pages either 1) have no dynamism, 2) implement all interactions and animations using CSS, or 3) use JavaScript primarily for transition effects (e.g., fade-in, lazy loading), and the core content can be fetched without executing it.

For the remaining 45.8% of pages, the statically crawled and dynamically crawled copies differ. Either the former is missing some content rendered in the page load, or some interaction on the page works on the latter copy but not the former. Thus, if a web archive solely uses a static crawler, it will often violate fidelity when serving archived pages.

Hence, I evaluate the utility of various approaches wherein a web archive uses a dynamic crawler for a small fraction of its crawls, which it then uses to fix the deficiencies of its static crawls.

4.2 Reusing resources across crawls

The first approach I consider for using dynamically crawled pages to benefit other static crawls is to reuse crawled resources across page snapshots. Recall that existing archives like the Wayback Machine already implement this strategy (§4.1); as part of serving a page snapshot crawled at a certain time, they serve any request for a resource using the copy crawled closest to that time. However, the impact of this practice in helping improve the fidelity of static crawls remains unquantified.

Note that reusing resources across a web archive's crawled pages differs from traditional web caching in three significant ways.

- First, a web cache can reuse resources only from a previous page load; a web archive can also reuse resources from later snapshots.
- Second, if an archive has no copies of a requested resource, it cannot go fetch the resource from the web; the resource may no longer exist or may have been modified.
- Therefore, it is okay for an archive to respond to a request for a specific resource URL with a previously crawled resource with a different, but similar, URL, e.g., due to CDN load balancing, the URL for the same resource can vary across loads [114]. I leverage pywb's in-built support for "fuzzy matching" of URLs [57].

4.2.1 Reusing resources from other pages

I begin by considering the utility of reusing resources across crawls of multiple pages. For each static crawl in my cross-sectional dataset, I evaluate the utility of reusing resources from dynamic crawls captured the day before or day after of other pages on the same site.

Figure 4.1(a) shows that, in comparison to reusing a resource only if the exact URL matches, fuzzy matching significantly increases reuse. Whereas, comparing Figures 4.1(a) and 4.1(b), I see that many static crawls which load with failed resource fetches do not violate fidelity. In both graphs, I see that the benefits of resource reuse improve with the availability of more dynamically crawled pages. Even a single dynamic crawl from a day before or day later can increase the number of statically crawled pages which can be served without violating fidelity from 54.2% to 82.8%.

Implication. When ordered by the frequency of the Internet Archive's crawls, for the median site in my dataset, the Wayback Machine hosts a median of 10 page crawls per day. If the Internet Archive were to use a browser for $\frac{1}{30}^{th}$ of its crawls – i.e., one dynamic crawl every three days for the median site – then every static crawl will be able to reuse resources from a dynamic crawl captured within a time gap of at most one day. My results suggest that this would suffice to serve over 82% of statically crawled page snapshots without any loss of fidelity.

4.2.2 Reusing resources from other times

On the flip side, reusing resources from other dynamically crawled pages is insufficient to fix the fidelity violations of *all* static crawls. Figure 4.1(b) shows that, even if an archive could reuse resources from as many as 40 dynamically crawled pages from the same site, it would be unable to address the deficiencies of 8% of static crawls. The primary reason is that many pages include dynamically fetched resources which are unique to them: either the page includes code for dynamic resource fetches which is not found on other pages, or the page's scripts read page-specific metadata from the HTML (e.g., page IDs) to initiate fetches.

A web archive will need to crawl each such page at least once using a browser. The resources on the page which are dynamically crawled can then be reused when serving either subsequent or prior static crawls of the same page.

Figure 4.1(c) shows that, even if the time gap between the static and dynamic crawls of a page is on the order of 10 weeks, reusing resources from the latter can significantly improve an archive's ability to serve the former copy without any loss of fidelity. For reference, on the Wayback Machine, I find that the median time gap between consecutive snapshots for the same page from any of the sites in my dataset is 79 days, i.e., roughly 11 weeks.

Implication. For 16.6% of pages, statically crawled copies violate fidelity even if one dynamic crawl from the same site exists a day apart. If the median page from this set is recrawled once every 10 weeks, then I recommend that 1 in every 3 crawls needs to be dynamic. Adding one-third of 16.6% to the aforementioned $\frac{1}{30}^{th}$ of the crawls of the remaining 82.8% of pages translates to the need for 8.9% of all crawls to be captured dynamically.

4.3 Accounting for inter-resource dependencies

My results from the previous section show that not all static crawls can be fixed by reusing resources from dynamic crawls. First, many pages include resources which are unique to each specific crawl of the page (e.g., a JSON listing the current prices of a set of products); these resources cannot be reused from any other crawl, even of the same page. Second, even if all the resources that are missing in a static crawl can be filled in using resources from other crawls, these replacements may not be consistent either with each other or with the statically crawled subset of the page's source. I address the latter problem in this section, and the former in the next section.

Resource bundling. To understand why replacing every missing resource independently can be problematic, consider the example in Figure 4.2. For a page on amazon.jobs, it shows how a modern JavaScript bundler like Webpack aggregates JavaScript source files and their dependencies into bundles. Common bundling usages include merging source files, resolving dependencies, minification, and tree-shaking (i.e., removing unused code).

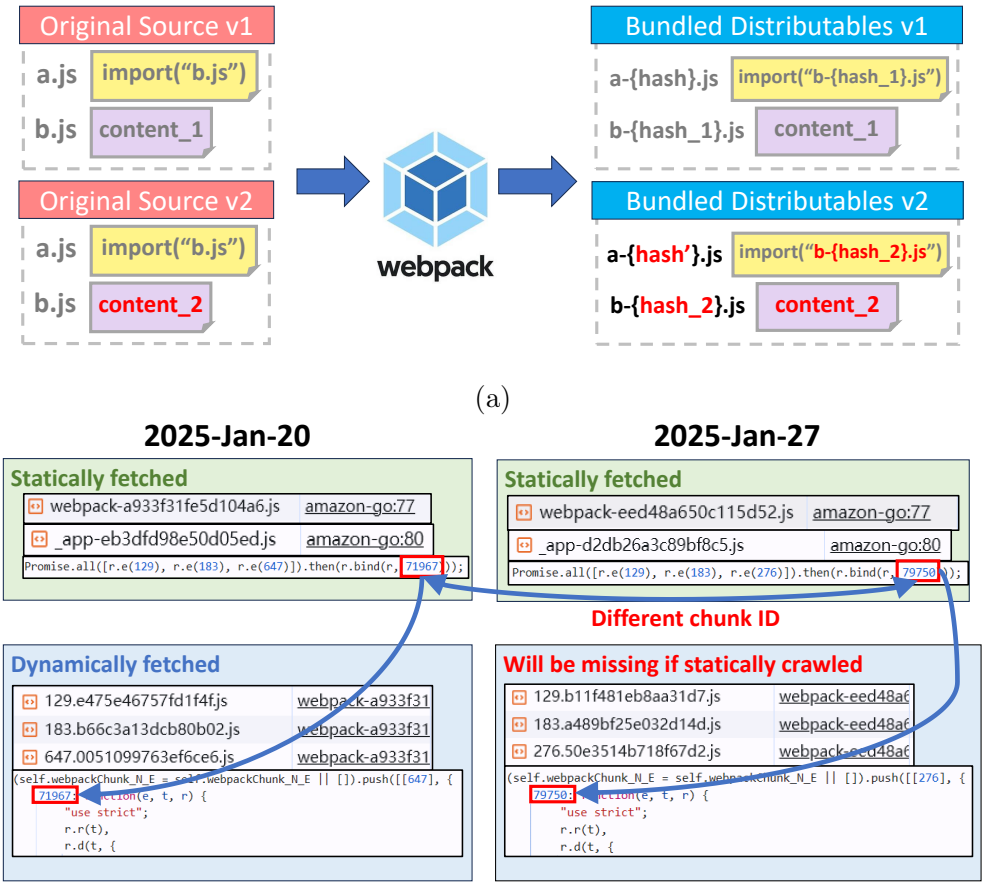


Figure 4.2: **(a) Demonstration of how bundling causes a JavaScript's URL to change when that file's content or of one of its dependencies is modified. (b) An example [45] wherein, after a week, rebundling results in new URLs for dynamically fetched JavaScript chunks and chunk ID mismatches between different bundle versions.**

As seen in the example, bundling often embeds content hashes or version identifiers into the filenames. pywb's fuzzy matching of URLs can reuse such resources despite the flux in their URLs. However, if it serves each missing JavaScript file from a different dynamic crawl, they can correspond to different versions of the bundle. Even if all the files missing in a static crawl are reused from one other dynamic crawl, they can be incompatible with the version number embedded in the page's HTML. Both of these will result in the broken load of a statically crawled page.

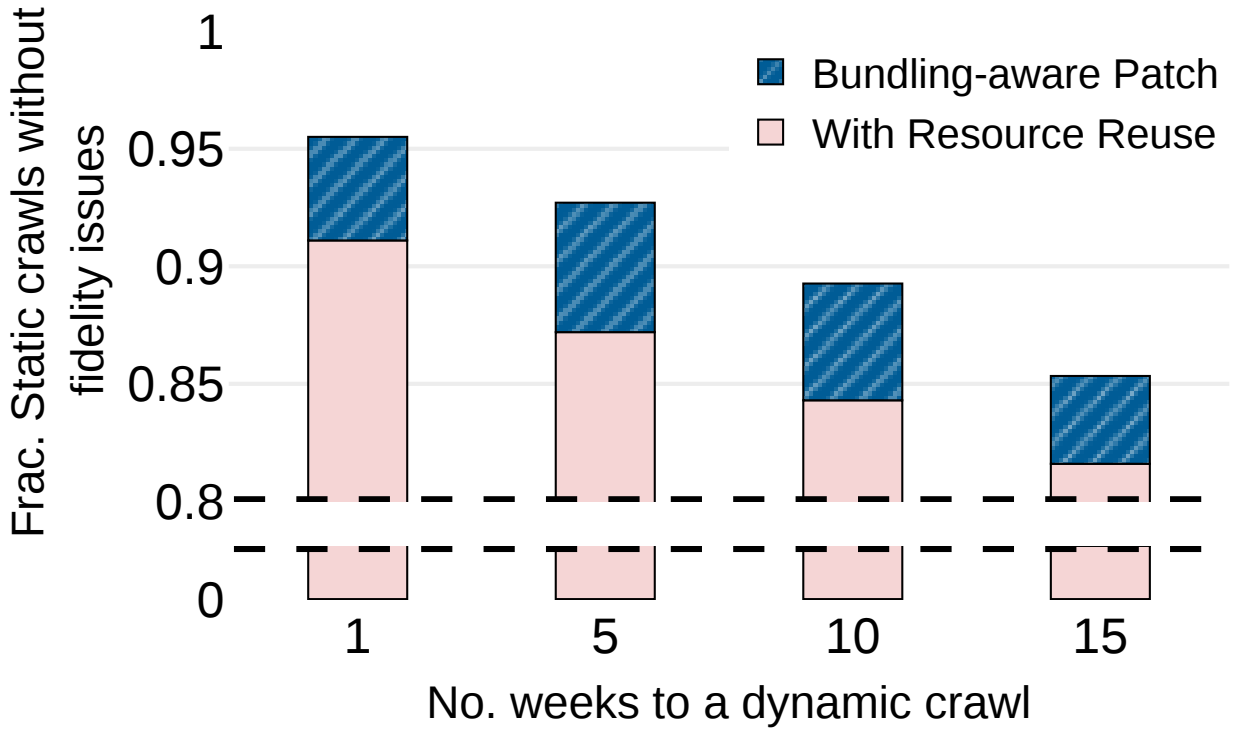


Figure 4.3: **Increase in the fraction of static crawls that can be served at high fidelity after bundling-aware patching.**

Bundling-aware patching. To prevent such incompatibility, I reuse the entire bundle of scripts from one dynamic crawl and modify the HTML of a statically crawled copy to be consistent with that version of the bundle. I will end up patching a statically crawled page snapshot using a bundle that differs from the one in use on the page when it was crawled. However, I observe that many of the bundles of a page's source remain functionally interchangeable. For example, changes may only involve bug fixes or feature updates which do not affect the page's rendering or functionality.

Specifically, I patch the HTML in any static crawl as follows. For each `<script>` tag in the HTML, I replace the script's URL with the URL found in the corresponding tag of the closest dynamic crawl of the same page. For instance, in Figure 4.2(b), I replace the `script` tag linking webpack-eed4xxx.js and _app-d2dbxxx.js with webpack-a933xxx.js and _app-eb3dxxx.js respectively from the dynamic crawl. In cases where the bundle's metadata is included in the `<script>` tags – e.g., `Next.js` [56] includes a `<script id="__NEXT_DATA__">` block in

the HTML, containing information about required chunks – I also replace these `<script>` tags correspondingly.

In my longitudinal dataset, I patch the HTML in this manner of any static crawl which has fewer scripts than the nearest dynamic crawl and includes script URLs from a different bundle version in the HTML. Figure 4.3 shows the impact as a function of the time gap between the static crawl and the dynamic crawl of the same page which I use to patch it. For example, consider the time gap of 10 weeks; recall that this is comparable to the median gap on the Wayback Machine for successive crawls of the same page from the sites in my dataset. The bar for 10 weeks in Figure 4.1(c) showed that, by reusing resources as described in the previous section, 84.3% of static crawls can be served without loss of fidelity. Patching the HTML to ensure that all missing scripts are served from the same bundle fixes the fidelity issues for an additional 7.2% of static crawls. However, my strawman patching also breaks 2.2% of static crawls which previously did not have any problems. These pages have fewer JavaScript dependencies, but the versions included on the page are critical for rendering. Using files from a different bundle version leads to fidelity issues. Overall, our patch achieves a net improvement of 5%, as shown in Figure 4.3. I defer to future work the task of extending my first attempt at bundle-aware patching to enable improvements without any regressions.

Implication. My findings highlight that JavaScript bundling-induced URL and content changes are a key barrier to reusing dynamically crawled scripts in static crawls. However, many scripts are still interchangeable across versions. By strategically patching the page's HTML to replace the entire bundle, I can eliminate a sizeable fraction of the fidelity violations observed with static crawls.

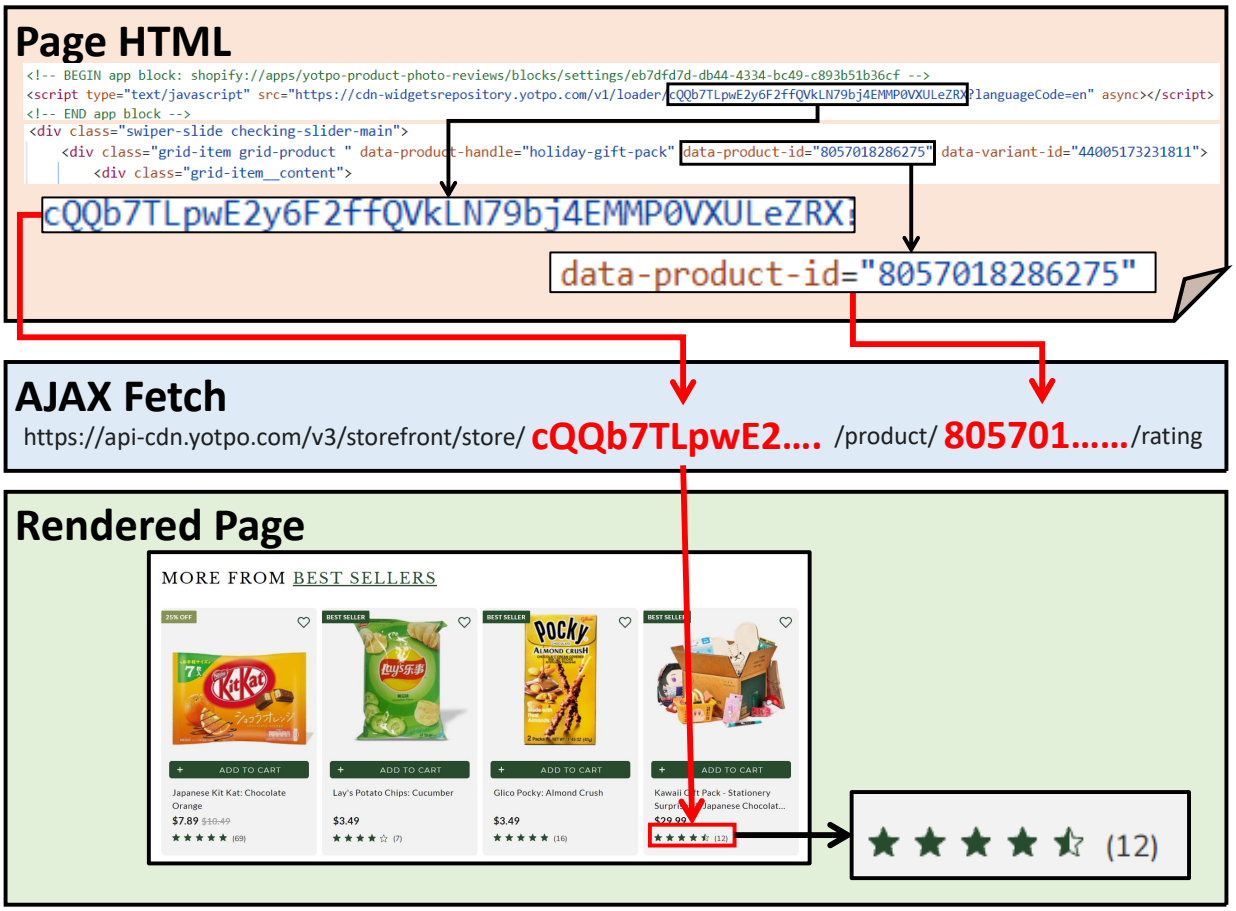


Figure 4.4: **An example page [54] on which the URL of an AJAX fetch varies across loads, but all the tokens in the URL can be traced to the same tags in the page's HTML.**

4.4 Handling resources which cannot be reused

Now, I turn my attention to the second reason why reusing resources across page snapshots cannot address all of the resources missing in static crawls. Many AJAX requests, which go unfetched by static crawlers, are unique to a particular page and the request's URL changes over time. The former precludes reuse of such resources from the dynamic crawls of other pages. Whereas, because of the latter, these resources cannot be reused across crawls of the same page when the gap between these crawls is large [135].

Construction of URLs of dynamically fetched resources. To overcome the inability to reuse such critical resources which are missing in static crawls, my key observation is that

the URLs of these resources can often be determined without executing JavaScript code. I find that the URLs of many resources fetched via AJAX requests are assembled using the content available in statically fetched resources. For instance, on the page in Figure 4.4, a dynamically fetched resource contains the current rating of a particular product. The URL of this resource is constructed using the tokens `"cQQb7T..."` and `"805701.."`, which are extracted from the `<script>` and `<div>` tags in the page's HTML. Though the URL of this resource can change across loads of this page, the URL in each load can be derived from the contents of the same tags in the corresponding HTML in that load.

Augmenting static crawlers. Therefore, an archive does not necessarily need to run a dynamic crawler in order to crawl all resources which are fetched during JavaScript execution. Instead, static crawlers can be enhanced to fetch these resources by using the content from the files they normally fetch. Prior work [124] has leveraged a similar approach for prefetching content in mobile applications.

I realize this approach as follows. Given a dynamic crawl of a page, I tokenize the URLs of all AJAX fetches, i.e., I split each URL into tokens using '/', '-', '.', etc. as delimiters. I then attempt to locate each token in other statically fetched resources on the page. If I can locate all the tokens of a specific dynamically fetched resource's URL, I construct a "recipe" of how to derive the URL based on where in the page's source the URL's tokens appear. In subsequent static crawls of the same page, the crawler can attempt to apply the same recipe and, if successful, fetch the corresponding resource URL.

Figure 4.5 shows that, after my enhancement, most of the static crawls in my longitudinal dataset which violated fidelity despite reusing resources from dynamic crawls of the same page are now fixed. In particular, Figure 4.5 shows that, when the time gap to the closest dynamic crawl increases, my enhanced static crawler can refetch resources which can no longer be reused. For example, the improvement in high-fidelity static crawls enabled by my augmentation of the static crawler increases from 6.3% to 10.5% when the gap to the closest dynamic crawl increases from 1 to 10 weeks.

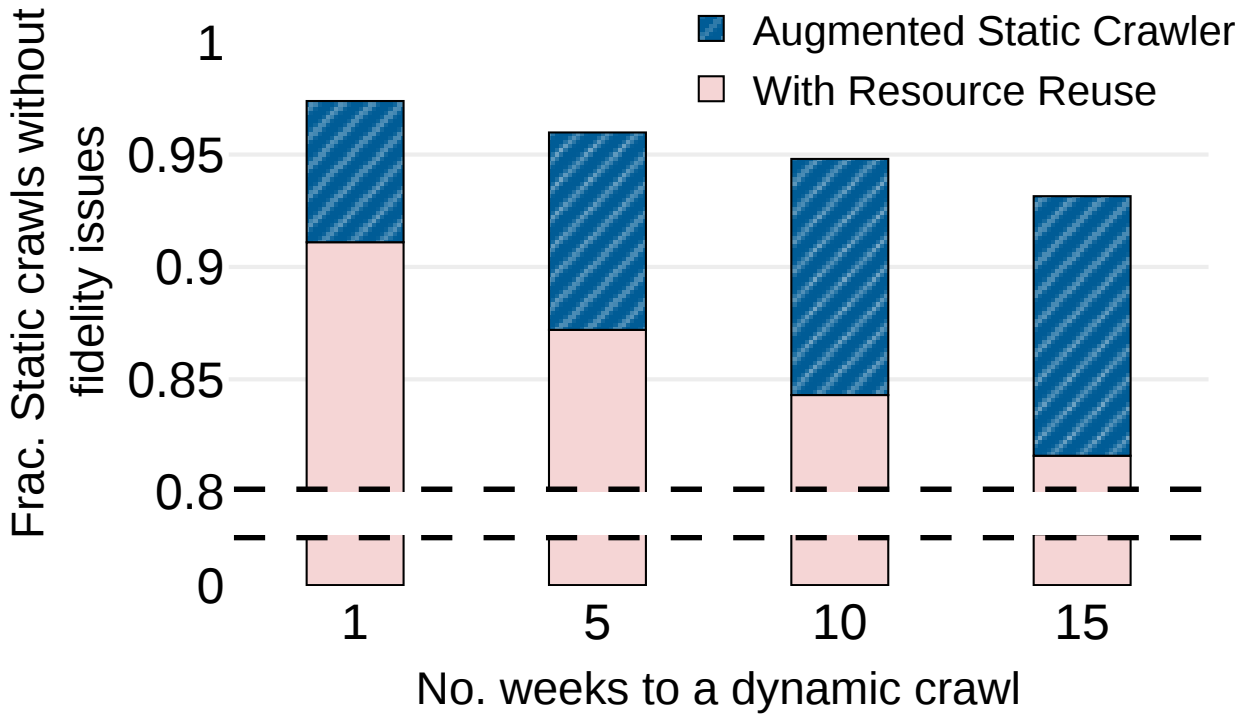


Figure 4.5: **Increase in the fraction of static crawls that do not violate fidelity if I augment a static crawler so that it can request dynamically fetched resources without executing scripts. Results are shown as a function of the gap between a static and the closest dynamic crawls of a page.**

The remaining static crawls which continue to suffer from fidelity issues are either because some dynamic fetches include non-deterministic URL tokens (e.g., the current timestamp) which cannot be inferred from the content of the page's source, or the required tokens are present in other dynamically fetched resources. Both of these limitations can be addressed, but I leave this for future work.

Implication. Despite the reliance on JavaScript for fetching resources, my results show that a) the URLs of such resources can be constructed from the content of the page's source, and b) the recipe for doing so is stable over the course of several weeks. Thus, using information from previous dynamic crawls, static crawlers can be augmented to issue AJAX requests without needing to execute scripts.

4.5 Efficiency–Fidelity Tradeoffs

To offer perfect fidelity, a web archive would have to crawl all pages with a dynamic browser-based crawler. As discussed earlier, this is however impractical, given the impact on crawling throughput. Here, I briefly compare each of the options available for improving throughput with respect to their impact on fidelity.

- Prior work [87] found that static crawling achieves 16x higher throughput than dynamic crawling. I showed that this preserves fidelity for only 54.2% of archived pages.
- Sprinter [87] combines dynamic and static crawling to offer a 5× speedup over dynamic crawling, while preserving 99.2% of the bytes fetched by a dynamic crawler.
- My analysis in this paper suggests that, by combining the approaches I proposed, an archive would need to crawl only 8.9% of pages dynamically. Using numbers from Sprinter's evaluation, I estimate that my work enables a 14.7× speedup over dynamic crawling, while sacrificing fidelity for 2.4% of archived pages.

I defer an empirical comparison of these options to future work.

4.6 Summary

To be able to capture and serve all pages with high fidelity, web archives have to today use a browser for all of their crawls, which is impractical at scale. By combining a range of simple strategies, we showed that it is feasible to instead make do with less than 10% of crawls being dynamic. I plan to work with the Internet Archive and other web archives to realize the new tradeoff between efficiency and fidelity that my work enables.

CHAPTER 5

Detecting and Diagnosing Errors in Serving Archived Web Pages[1]

In the previous chapter, I described how to archive as many necessary resources as possible to preserve the fidelity of archived copies while maintaining high crawling efficiency. However, I found that even when all required resources are available, archived copies can still exhibit fidelity issues in users' browsers. The root cause often lies in archival systems incorrectly serving these archived copies.

In this chapter, I highlight the challenges that web archives face in accurately replaying archived copies. With the emergence of JavaScript, archival systems need to rewrite JavaScript APIs in the browser to ensure that all AJAX requests are rerouted to the archive's host. However, the diversity and complexity of modern JavaScript make this task difficult. As a result, on many modern pages, an archive's edits to crawled scripts result in a loss of fidelity, i.e., an archived copy fails to accurately mimic the original page even when the archive has crawled all resources on the page.

To help the developers of archival systems identify and fix the bugs that result in incorrect rewrites of crawled pages, I present FidEx. First, FidEx enables accurate identification of the pages on which an archive violates fidelity. It does so by tracking and comparing

[1]This work also involves Huanchen Sun from the University of Southern California. In this work, I led the end-to-end design and implementation of FidEx, performed all evaluations aside from those involving manual analysis, worked with pywb developers to develop and integrate fixes, and contributed the majority of the paper's writing.

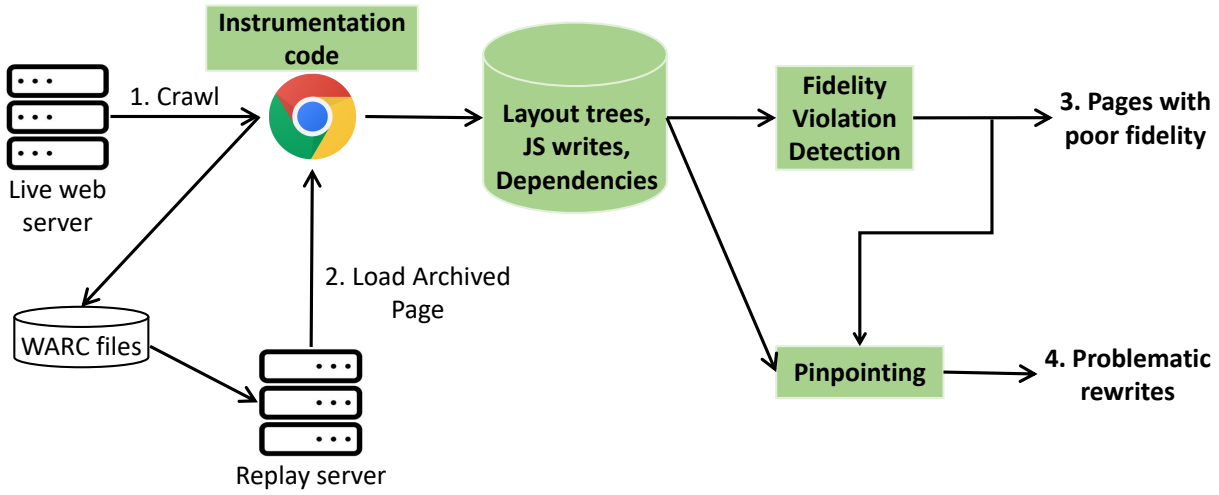


Figure 5.1: **A web archive's workflow augmented with** FIDEX**,** whose components are highlighted in green.

the execution of scripts between when a page is crawled and when its copy is loaded. In comparison to existing methods that compare the two loads using either screenshots or the errors reported by the browser, FIDEX reduces the false positive rate from around 70% to less than 10%. Second, on every page on which it identifies a loss of fidelity, FIDEX pinpoints which subset of the archive's edits to the page are erroneous. Leveraging this input to fix bugs in the most widely used archival system, I reduced the fraction of archived pages which violate fidelity from 15% to 9%.

5.1 Overview

To help web archives detect and fix their incorrect rewrites of scripts, I enhance their workflow with FIDEX. As illustrated in Figure 5.1, FIDEX augments an archive's browser-based crawler. Both when an archive crawls a page from the web and when it loads its local copy to check the quality, the FIDEX-augmented browser records fine-grained information characterizing the page load. FIDEX compares the logs gathered during the two loads to identify a mismatch between the original page and its archived copy. On every page on which it identifies a loss of fidelity, FIDEX pinpoints the erroneous rewrites which need to be fixed.

5.1.1 Goals

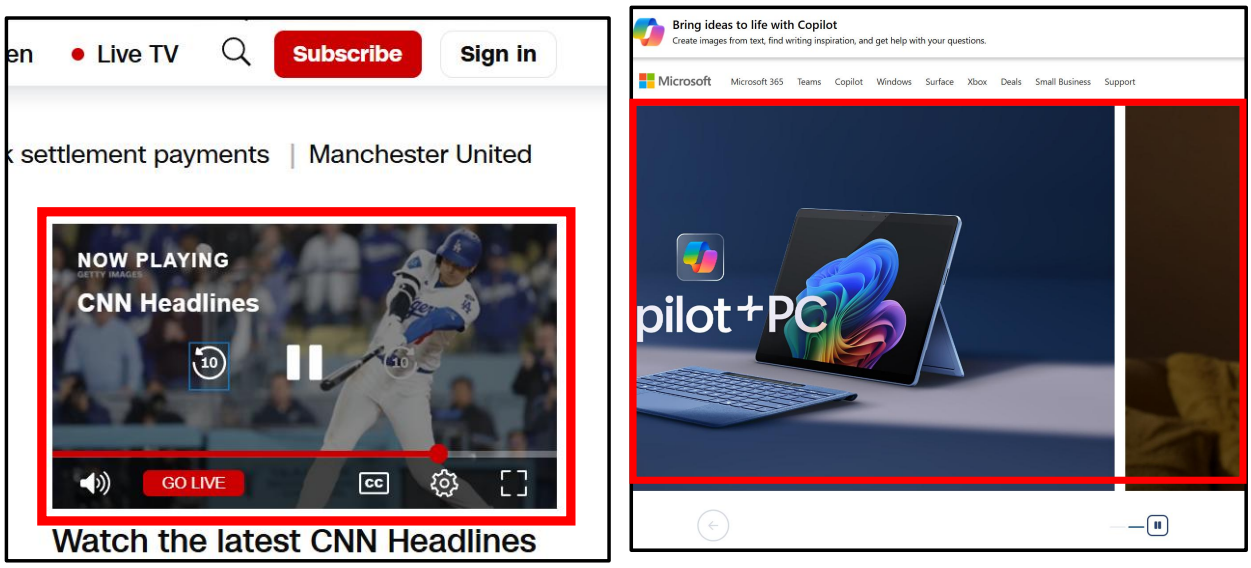
I design FIDEX with three objectives in mind.

- **Accurate detection:** First, I strive to flag fidelity violations with high accuracy, prioritizing the need to minimize false alarms. Whenever an archived page fails to precisely mimic the original page, an assessment of the severity of the differences is ultimately subjective; it depends on what page content/functionality users and developers care about. I aim to inform this assessment by highlighting the differences and estimating their potential impact, e.g., the size and position of the elements which are missing on the archived page.
- **Accurate diagnosis:** Second, FIDEX should pinpoint only the rewrites which indeed cause observed fidelity violations. In doing so, I hope to simplify the task of human developers to identify the bugs in archival software that need to be fixed.
- **High-throughput crawling:** Lastly, I aim to preserve an archive's ability to crawl pages at high throughput. Therefore, FIDEX's modifications to browser-based crawlers must impose low overhead.

5.1.2 Challenges

Fidelity violations vs. page dynamism. A common approach to identify differences between loads of a live page and its archived copy is to compare screenshots [58]. However, on pages such as Figure 5.2(a) which include videos and GIFs, the screenshot can vary across loads based on when the screenshot is captured.

To compare multiple loads of a page in a more robust manner, prior work [78, 77, 137] has therefore proposed extracting and comparing a structured representation of the page's appearance. However, no representation of the content in each visual component suffices to accurately account for the dynamism on modern web pages. For example, consider the page in Figure 5.2(b) which contains an image carousel. If I represent the bounding box within which the carousel is contained by the the resource which provides its content [78, 77], then



(a) (b)

Figure 5.2: **(a) An animation on cnn.com can cause screenshot to differ between live page and archived copy. (b) Image carousel on microsoft.com can cause page layout to also differ.**

two loads will appear to differ if the image shown in the carousel at the time of comparison differs. If I instead use a more coarse-grained representation – e.g., that *some* image is displayed within the bounding box [137] – then, an archived page may match the original page even if the carousel is rendered dysfunctional due to the archive's incorrect rewrite of the page.

Runtime errors $\not\Rightarrow$ fidelity violations. Once a developer confirms the loss of fidelity when loading an archived page, it would be natural for them to focus on the errors (e.g., failed resource fetches or runtime exceptions) observed only when loading the archived copy, but not when the copy was crawled. This would be akin to prior work on fault localization (FL) [146], which leverages crash stack traces [128] or bug reports [140] to pinpoint the root cause of a fault. However, in my setting, such an approach can lead developers astray as errors that occur when loading an archived page often have no impact on the page; many scripts either do not visibly affect the page (e.g., scripts related to analytics or performance monitoring) or power functionality that will not work on archived pages (e.g., logins, live

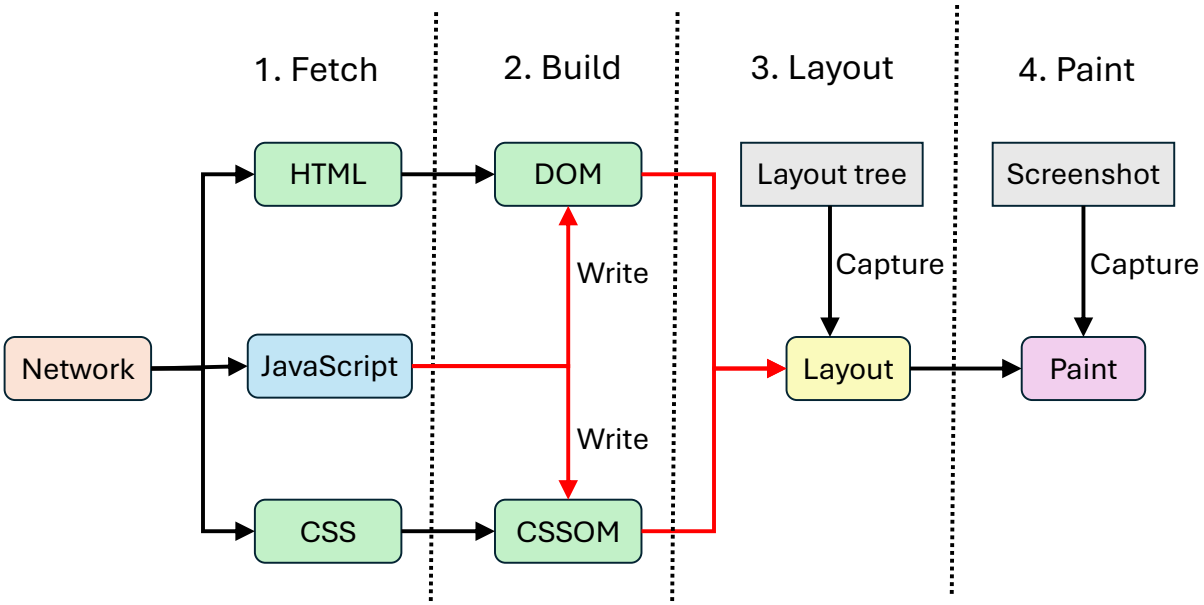


Figure 5.3: **Illustration of browser's critical rendering path.**

chat, and reCAPTCHA).

Fidelity violations $\not\Rightarrow$ runtime errors. A final challenge is that many archived pages violate fidelity without the browser reporting any errors. Scripts on the page might catch and handle errors without making them externally visible. Or, incorrect rewriting can lead to divergence in the control flow without triggering any exceptions. Prior work [93, 63, 118, 138] attempts to locate such silent errors by identifying code which is executed only on test cases which fail, but not on successful tests. However, such methods are ill-suited for my context. If I compare the correct load of a page from the web to the incorrect load of its copy from an archive, the difference between the two loads will correspond to *all* of the archive's instrumentation of the page's code, not just the subset responsible for the loss of fidelity.

5.1.3 Guiding observations and approach

My approach to address the aforementioned challenges is guided by how modern browsers render pages (Figure 5.3). After the browser fetches a page's resources over the network, it constructs the DOM and CSSOM trees, and combines them into a layout tree [10] which

determines the positions and dimensions for every visible element. The browser then paints pixels for each element in the layout tree.

Viewing a page load as in Figure 5.3 makes it apparent why using a screenshot or the page's layout is insufficient to compare two loads of a page. The DOM and CSSOM trees, though initialized from the page's HTML and CSS stylesheets, are continually updated by JavaScript code on the page. Therefore, when comparing an archived copy to the original page, I cannot compare any particular snapshots captured when loading the two versions; that comparison is prone to *when* the snapshots were captured.

Given the dynamism on modern pages, I instead declare that an archive preserves fidelity when serving its copy of a page if, for *any* screenshot captured when the page was crawled from the web, the archived copy will *eventually* produce an identical screenshot. I assume that the archive ensures that any sources of non-determinism encountered when crawling a page are replayed when serving the page [86]. For any archived page, I separately evaluate the fidelity of its initial load and of each of the interactions (e.g., hover, click) on the page.

To evaluate whether any archived page satisfies the criterion defined above, FIDEX tracks the JavaScript writes which affect the page's visual appearance. I do so both when crawling a page and when the crawled copy is later served. On the one hand, it is problematic if any of the JS code which contributed to the page's rendering when it was crawled is never executed on the archived copy; whether the page's layout matches up is immaterial. On the other hand, even if the page's end state differs, I can reliably infer that fidelity was preserved if the code that contributes to the page's visual appearance is identical in both loads.

Tracking the execution of scripts also enables FIDEX to aid the developers of archival systems in two ways. First, by establishing how the execution of every script on a page impacts the page's rendering, it can precisely determine which runtime errors causally lead to observed fidelity issues. It, thus, frees developers from having to inspect irrelevant errors. Second, FIDEX's comparison of how the scripts on a page are executed prior to and after rewriting enables it to identify discrepancies which contribute to differences in the page's

appearance. Consequently, FIDEX can shed light on incorrect rewrites which are otherwise not apparent from browser-reported errors.

5.2 Design

To realize the approach described above, I need to answer several questions. What information should FIDEX collect during page loads? How should it use this information to accurately and comprehensively identify fidelity violations? What methodology should FIDEX employ to nail down which of the errors observed when loading archived pages are worth digging into? How can FIDEX uncover problematic rewrites which do not result in any errors? I describe my answers to these questions next.

5.2.1 Identifying fidelity violations

FIDEX represents the outcome of any page load as the page's layout tree along with the set of JS writes executed on each node in the tree. The layout tree is an internal browser structure that cannot be directly accessed. But, I approximate it by running JavaScript that iterates through the DOM tree to identify all visible elements, and collects relevant rendering details. I gather every element's dimensions and computed CSS styles, such as animation properties. For any element containing content, such as `<img>`, `<p>`, and `<video>`, FIDEX also records attributes that affect the browser's composition and painting of the element.

FIDEX captures JS writes by overriding a) browser Web APIs that modify DOM elements (e.g., Node's `appendChild`, Element's `setAttribute`) and b) property setters for `CSSStyleDeclaration` and `ClassList`. FIDEX associates each write with the element on which the operation is invoked, all descendant elements in the layout tree, and any elements included in the argument list. FIDEX records only those JS writes which modify the associated element's layout information.

A tricky, but important, detail is to determine what constitutes a unique *write*. On the

```
let changed = False;
// Impl 1: Change element's class upon message recv
window.addEventListener("message", function(event) {
    changeElementClass(element, event);
    changed = True;
});
// Impl 2: Fallback if no message within 10 seconds
window.setTimeout(function() {
    if (!changed)
        changeElementClass(element, null);
}, 10000);
```

Figure 5.4: **Example inspired by the code found on skype.com in Dec 2024,** wherein a fallback implementation runs if the primary one fails, both resulting in the same layout outcome.

one hand, in different loads of the same page, the JS code on the page might execute the same API with different arguments. For example, a carousel might periodically switch the displayed image by modifying the `src` attribute of an image tag. On the other hand, I cannot identify a write simply by the API invoked. This will merge multiple invocations of popular APIs (e.g., `appendChild`) into a single write. FIDEX balances these considerations by representing every unique write by the combination of a) the API invoked and b) the call stack at the time of invocation.

5.2.1.1 Comparing JavaScript execution

Given the layout trees from the loads of a live page and its archived copy, and the set of JS writes captured for every element in the trees, FIDEX faces two options for how to compares the two loads.

- On the one hand, if I only match JS writes, I find that I falsely flag pages wherein, to maximize compatibility across browser environments, web developers employ multiple implementations (like in Figure 5.4) to achieve the same functionality. In such cases, loads of a live and archived page might execute different code, but produce the same layout tree.
- On the other hand, even if the JS writes for some elements differ, say I consider an archived page as equivalent to the original live page if the layout details of every visual element

match. Then, I risk failing to detect instances of broken dynamism. For example, as previously discussed in §5.1, on a page which contains an image carousel, though the layout of the live and archived pages may match, the carousel may be dysfunctional on the archived page.

In this tradeoff, I choose the latter option for two reasons. First, I empirically observe that the reduction in false positives outweighs the potential increase in false negatives. Second, I find that the risk failing to detect broken dynamism on an archived page can largely be mitigated by waiting for the "working set" of dynamic behaviors to complete before capturing the layout tree (§5.2.1.2).

Specifically, FIDEX compares a live page to its archived copy as follows. FIDEX traverses the layout trees from both loads from the root. For elements in the same position in either tree, FIDEX considers the two elements as equivalent either a) if both elements include the same layout information, or b) if the set of JS writes associated with these elements is identical. If all elements are matched during the traversal, FIDEX concludes that the live and archived pages are equivalent. Otherwise, FIDEX flags the archived copy as having fidelity issues.

5.2.1.2 Timing of comparison

Thus far, I have discussed what information FIDEX records when loading pages and how it compares this information to identify fidelity violations. But, *when* in a page load can FIDEX be certain that it has collected all the information necessary for comparison? This is an important consideration for two reasons. First, some JS-controlled dynamic components (e.g., carousels) kick into action only a few seconds after the browser's `onLoad` event fires. Second, when crawling a page and later testing its archived copy, it is important to trigger all interactions on the page, as some of these result in extra network fetches. FIDEX must compare the two page loads only after the changes associated with the fetched resources have been applied.

Therefore, it is essential for FIDEX to wait until all dynamic components on the page exhibit their intended behavior before collecting the layout tree and the corresponding JS writes. I observe that the key source of delayed dynamism is asynchronous JavaScript execution. For example, carousels are often implemented by adding a callback through `setInterval` or `setTimeout` to modify the corresponding elements. Similarly, interactions that involve additional resource fetches often attach a callback to fetch APIs that interact with the DOM.

To wait to see all dynamism on a page, FIDEX overrides Web APIs that add asynchronous callbacks. The target APIs include timers (e.g., `setInterval` and `setTimeout`), promise fulfill/reject callbacks (e.g., `Promise.then` and `Promise.reject`), and network request callbacks (e.g., `XMLHttpRequest.send`). Whenever any of these APIs are invoked, FIDEX adds the provided callback to its unfinished list. FIDEX removes a callback from the list once it is fired, and waits for the list to become empty. To account for dynamic components which, in order to change the page periodically, are implemented with callbacks that repeatedly add themselves to the event loop, FIDEX adds every callback function to the list at most once. Additionally, FIDEX times out on callbacks associated with promises that never resolve (i.e., remain in a pending state forever).

5.2.2 Pinpointing root cause of fidelity violations

Next, I shift my attention from detection to diagnosis.

FIDEX aims to aid developers in their effort to identify the bugs in archival systems which result in fidelity violations. First, FIDEX pinpoints the runtime errors which result in either missing content or broken functionality. Prior work [128, 144] has shown that crash stack traces help developers identify code defects and fix bugs. Second, on pages which violate fidelity without resulting in any errors, FIDEX highlights the rewritten code on the page – a specific line, function, or file – which results in the observed fidelity violation. FIDEX may not always identify the precise origin of divergence between a live and archived page. But,

by pointing out problematic rewrites, it will help developers focus their debugging efforts.

5.2.2.1 Runtime error based diagnosis

Runtime errors in JavaScript are caused by 1) syntactically incorrect script files that cannot be parsed, or 2) uncaught exceptions that halt execution. In the former case, all code in the script that cannot be parsed does not get executed. In the latter case, the browser skips executing the remaining code in the script file where the error occurs. As a consequence, the archived page ends up missing many modifications to the layout tree that were made in the load of the live page.

Detecting runtime errors when loading a page is straightforward, e.g., with Chrome, FidEx just needs to listen for the `exceptionThrown` event from the browser's DevTools protocol. If a Syntax error is thrown only when loading an archived page, it is evident that the script where the error occurs has been incorrectly rewritten. Similarly, a Type error is often thrown on an incorrectly rewritten line of code.

However, not every runtime error caused by rewriting will result in fidelity violations. For example, a syntax error in a script that is fetched but never used has no effect on the page's rendering. Similarly, exceptions thrown in asynchronous events may not halt any JS writes that impact the layout tree.

A straightforward way to validate if a runtime error impacts page fidelity would be to reload the page from the live web and inject the same error in that load. Then, I can check whether the differences between the live page and its archived copy reduce. However, the live page on the web will likely differ from the version originally crawled; the page may even no longer exist. Loading the archived copy in proxy-mode (§2.1.2), which is unaffected by the archive's rewrites, is also not always an option since not every archival system supports this. Moreover, whether I inject errors into live pages or archived pages in proxy-mode, reloading every page once for each runtime error observed on the page will impose significant overhead; in my dataset, I observe 2.1 additional errors on average per archived page which violates

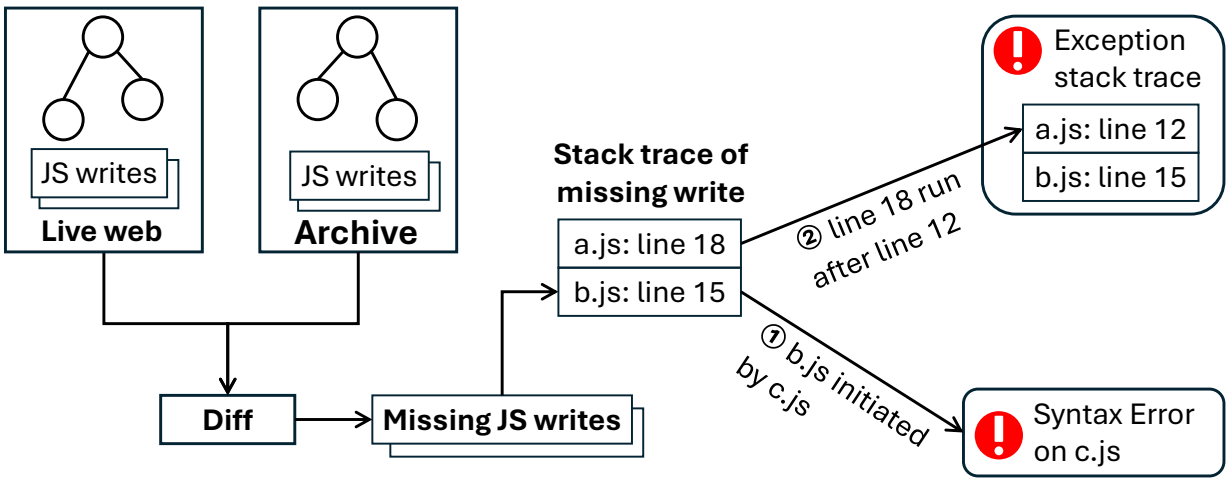


Figure 5.5: **How FIDEX confirms whether a runtime error led to violation of fidelity.** Either syntax errors (case 1) or uncaught exceptions (case 2) can cause missing JS writes.

fidelity.

Therefore, FIDEX attempts to evaluate the impact of every runtime error by correlating it with the JS writes missing in the load of the archived page. Recall that FIDEX's comparison of layout trees from the loads of the original and archived pages reveals the JS writes that were executed in the former load but not the latter. FIDEX compares a runtime error with each such missing JS write to determine whether that write was not executed because the error prematurely terminated the execution of the script in which the error is thrown.

FIDEX makes this determination by comparing the stack trace of the runtime error with the stack trace of each missing JS write. FIDEX determines that the error is the cause for the missing write if any frame in the error's stack causally precedes any frame in the write's stack. As shown in the example in Figure 5.5, FIDEX tracks two kinds of precedence. On the one hand, within the same scope, an earlier line causally precedes a later line. On the other hand, if a script initiates the fetch of another script, then all lines in the former prior to the fetch precede all the code in the latter. FIDEX tracks these dependencies when a page is crawled. When it later evaluates the quality of the archived copy, FIDEX transitively applies these stored dependencies to determine which of the new runtime errors causally precede

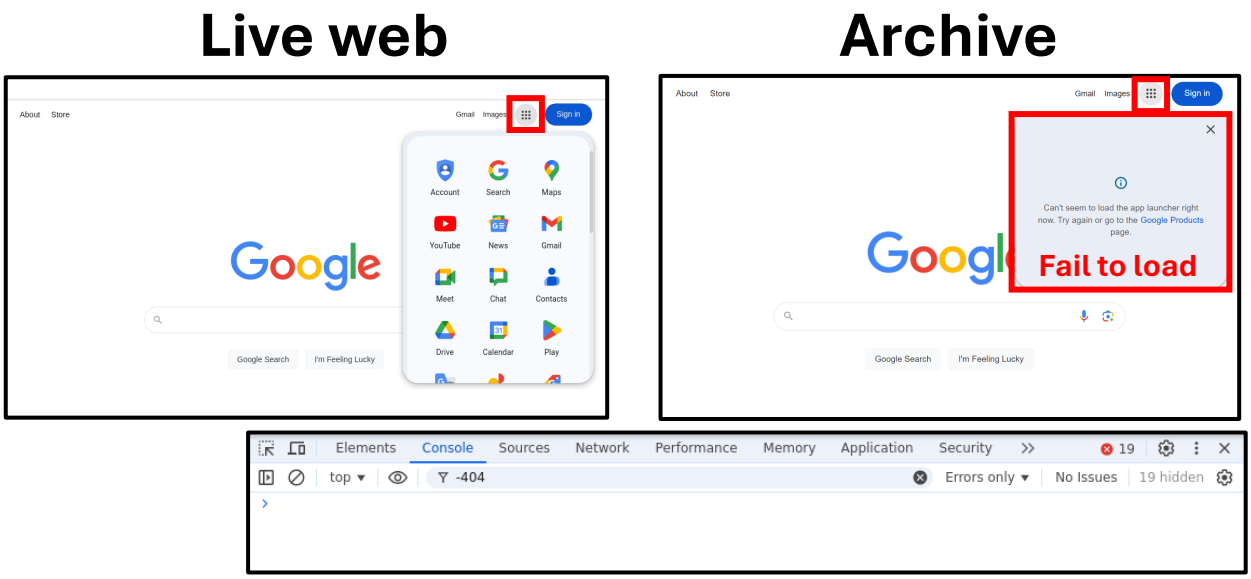


Figure 5.6: **On the archived copy of Google's homepage, the Google Apps drop-down fails to load when the associated icon is clicked, even though no error messages are shown in the browser console (failed fetches are filtered out).**

any missing JS write.

5.2.2.2 Detect impactful execution divergence

Erroneous rewriting can also lead to fidelity issues without resulting in any runtime errors; Figure 5.6 shows an example. When exceptions are caught and handled by `catch` blocks or rewritten functions return incorrect values, though execution continues, it diverges from expected behavior.

Discover silent errors. Unlike runtime errors, it is not straightforward to detect errors that are not exported by the browser. First, FIDEX's tracking of the flow of execution, in order to identify when the loads of the original page and its archived copy diverge, cannot introduce large overheads. For instance, using the browser's debugger to set a breakpoint on every caught exception would significantly slow down crawling of pages. Second, I can keep FIDEX's overhead low by having it track the execution of only those functions that are dynamically rewritten; in my setting, any divergence in the execution of scripts must stem

```
1  const originalWindow = window;
2  // Override of the getter for property "prop" on element "target"
3  function overrideWindowGet(target, prop) {
4      Erroneous overrides ...
5  }
6  window = new Proxy(window, {
7    get: function (target, prop) {
8      const retval = overrideWindowGet(target, prop);
9      if (proxyToObj(retval)!== originalWindow[prop])){
10       console.log("Spec violation for window.get)
11     }
12     return retval;
13 })
```

Figure 5.7: **Example of FIDEX detecting silent errors.** Code highlighted in green is added by FIDEX to check if overridden APIs satisfy their specification.

from the archive's rewrites. Thus, FIDEX wraps every overridden function in a try-catch block to detect when its incorrect execution results in an exception that is caught outside the function.[2] However, if an overridden function does not result in an exception, it is hard to identify when it executes incorrectly; after all, the purpose of rewriting a function is to modify its behavior.

I observe that dynamic rewrites of web APIs [16] only intend to ensure that any dependent resource fetches are directed to the archive. The overridden APIs must still satisfy their specified interface. For example, during the load of an archived page, the `Window` object needs to be overridden using JavaScript Proxy to ensure that the resources added to `Window` are fetched from the archive. Despite this, the getter for `Window.parent` must satisfy its specification of enabling access to the parent window of the current frame (if available). The only impact of the `Window` object being overridden ought to be that the `Window.parent` getter should return a proxified version of its parent frame, rather than the original.

Hence, as shown in Figure 5.7, FIDEX discovers silent errors by adding specification checks to overridden functions. FIDEX attaches to every overridden (proxified) object a reference to the corresponding native object. For values returned by rewritten getters, FIDEX checks that

[2]Typically, archival systems embed into every page a single script (e.g., wombat.js) which contains all the overridden Web APIs.

the attached native object still follows the API specification. Overrides on setters or fetch-related APIs are harder to verify. I have found some common issues include setting incorrect values (e.g., setter on iframe's `srcdoc` is incorrectly scoped) or returning incorrect values (e.g., fetched JSON files are identified as JavaScript, and incorrectly rewritten). FIDEX also adds corresponding checks for them.

Validate impact of silent errors. Unlike runtime errors, silent errors do not halt execution, and may have no impact on the page. For example, failing to get the correct parent frame will prevent the handler for cross-frame messages from being triggered in reCAPTCHA. But, its failure has no impact on the user.

Therefore, FIDEX has to validate the effect of every silent error by comparing how the page looks with and without the error. FIDEX does so by selectively reverting the overridden function/object in which the error is observed to its original implementation. FIDEX reloads the page after this modification. If the number of differences in the layout tree between the original page and the archived copy reduce, FIDEX can confirm that the silent error is indeed a cause for the violation of fidelity.

5.3 Implementation

My implementation of FIDEX includes ~5.8K LOC in Python and ~3.1K LOC in JavaScript.

Browser-based crawling. Browser-based crawlers used by web archives, such as Browsertrix [48], have substantial performance drawbacks [87]. Therefore, I implement a relatively lightweight browser-based crawler which uses Webrecorder, a Chrome extension for recording web pages.

Tracking JavaScript writes. FIDEX tracks JavaScript writes by overriding web APIs that modify DOM nodes, styles, and classes. FIDEX uses Puppeteer [9] to automate browser-based page loads, and leverages Puppeteer's `evaluateOnNewDocument` method to inject the overridden implementations right after any document is initialized. By doing so before

archival systems override these APIs, FidEx ensures that it tracks native calls to APIs that adhere to their specifications. To prevent FidEx's instrumentation from interfering with the archive's rewrites, I only record the target element and arguments when overriding any method, but never modify them.

Relating runtime errors to missing JS writes. FidEx uses Esprima [5] and BeautifulSoup [2] to parse JavaScript from each stack frame into an abstract syntax tree (AST). FidEx strips off rewrites added by the archival system to align the AST between the live and archived pages. Then, FidEx uses static analysis on aligned ASTs to check for precedence relationships between stack frames.

Maintainability and generalizability. To ensure that FidEx is easy to maintain as web APIs that write to the DOM and CSSOM evolve, I categorize all such APIs into two types: APIs that write by setting an element's attributes, and APIs that write by calling an element's methods. FidEx implements a general overriding function for each type. To track any new API, I only need to add it to the list of APIs for the overriding function of the corresponding category.

FidEx will need to be modified to make it compatible with different archival systems as they vary in how they rewrite scripts. It will also be necessary to add new specification checks to detect silent errors. However, most of FidEx's implementation is independent of precisely how and what a specific archival system chooses to rewrite. For example, instrumenting page loads to capture layout trees and JS writes is only dependent on the browser used. The code for comparing layout trees can also be reused.

5.4 Evaluation

I evaluate FidEx from three perspectives: 1) its ability to correctly flag pages that exhibit fidelity issues, 2) its ability to pinpoint the erroneous rewrites that lead to fidelity violations, and 3) its efficiency in all aspects of its operation. Rather than focusing on any one archive,

I evaluate FidEx on version 2.9.0 (the latest version in Sep 2025) of pywb [17], the software most widely used by web archives [8] to statically and dynamically rewrite references to resources in crawled pages. To examine FidEx's generalizability, I also extend it to work with an alternate system for serving archived web pages, ReplayWeb.page [18].

The key takeaways from my evaluation are:

- Out of 80K pages sampled from sites in the Tranco list, FidEx finds that the archived copies of 15% of pages violate fidelity, with most of these issues contributing significantly to the page's visual content. FidEx's precision in identifying fidelity violations is 90%, whereas relying on screenshots or browser-reported errors for the same purpose results in a false positive rate of at least 65%.
- FidEx pinpoints the erroneous rewrites on 64% of the pages where it detects fidelity violations. 30% of the pinpointed errors correspond to cases where execution diverges without resulting in any exceptions.
- The top 4 most common errors pinpointed by FidEx account for > 60% of all pinpointed errors. I worked with the developers of pywb to fix the bugs in pywb which result in these errors. These bug fixes reduced the fraction of archived copies with fidelity violations from 15% to 9%.
- FidEx reduces crawling throughput by less than 5%, which is lower than the overhead added by taking screenshots when crawling pages.

5.4.1 Dataset

To gather a representative set of pages from the web, I start by sampling 20K sites from Tranco [122]: 2K from ranks [1–10K], 3K from ranks [10K–100K], 5K from ranks [100K–500K], and 10K from ranks [500K–1M]. For each site, I query the Wayback Machine and select at random 4 non-home pages that exist today. In combination with the homepages, this gives me 100K pages in total.

I then run FidEx with 3 loads for each page, triggering up to 20 interactions on the page in

each load.[3] First, I record each page from the live web. Second, I load every page's archived copy using pywb's proxy-mode and filter out pages which violate fidelity for two reasons: 1) because of issues for which solutions exist but have not been incorporated into Webrecorder (e.g., record sources of non-determinism in order to replay them when serving pages [86]), or 2) due to implementation issues in pywb (e.g., pywb does not support WebSockets). This leaves me with 83K pages, of which I sample 80K for my study. Finally, I load the archived copies of these 80K pages from pywb's replay server.

5.4.2 Fidelity violation detection

First, I evaluate FidEx's ability to detect fidelity violations. I compare it to three baselines.

- *More Errors.* In the first baseline, I log all runtime errors and declare that an archived copy violates fidelity if any additional errors are observed when loading it, compared to the errors seen when the page was crawled.
- *Screenshot.* Akin to Browsertrix's quality assurance [58], I compare the load of an archived page to its original crawl using screenshots.
- *Layout Tree.* Similar to prior work [78, 77] which uses a structured representation of every page, I compare the layout trees between a crawled page and its archived copy.

In all cases, I compare the original page and its archived copy both after the initial load and after every interaction. For every baseline, I record its chosen representation of the page at the same time when FidEx does so (§5.2.1.2).

As shown in Table 5.1, FidEx determines that 12K (i.e., 15%) of the 80K archived pages violate fidelity. About two-thirds of these correspond to visual mismatches between the original page and its archived copy. In the remaining, some functionality on the page that ought to work on the archived copy is broken. There are more cases in the former category

[3] I group interactions by their class names and trigger one each from the first 20 groups. I find that 80% of pages have less than 20 interactions.

	More Errors	Screenshot	Layout tree	FIDEX
Positives	34,200	55,392	14,094	12,019
Negatives	45,800	24,608	65,906	67,981

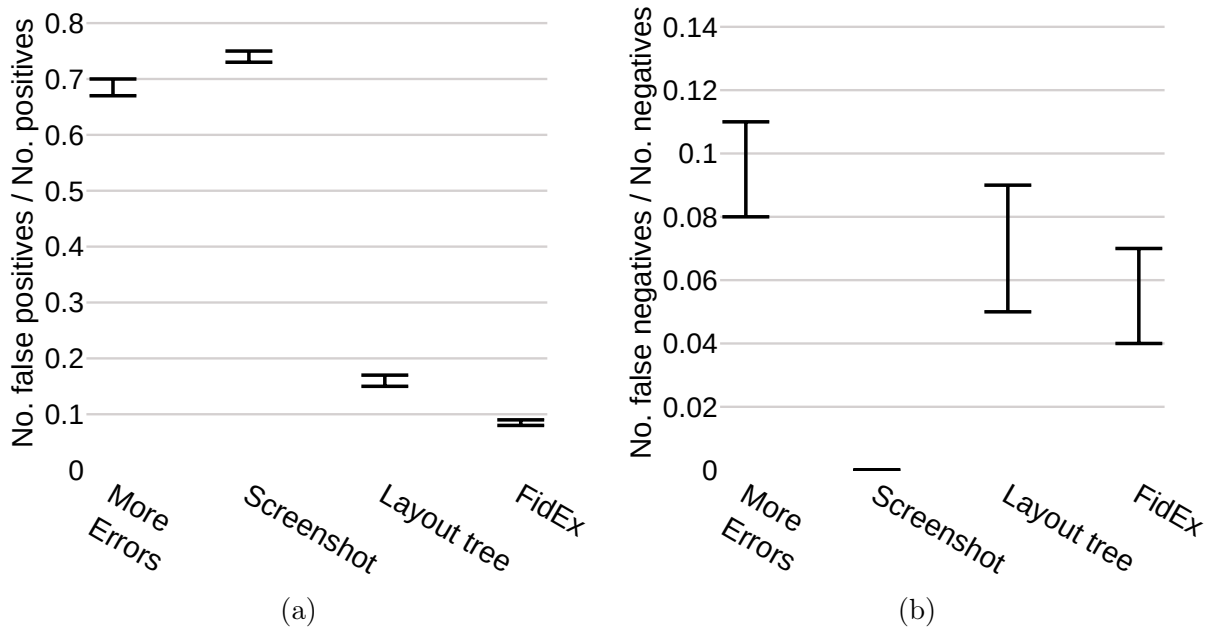
Table 5.1: **No. of positives (detected fidelity violations) and negatives classified by each approach out of 80K archived pages.**

because, when pywb's rewriting of a page's source breaks some interactions, it typically also affects the initial page load. For instance, many pages include a hamburger icon which, when clicked, displays a menu bar. On pages where pywb's rewrites render such clicks dysfunctional, I find that the icon often does not even get displayed.

The number of pages that FIDEX deems violate fidelity is lower compared to all other baselines because of its better accuracy. Given the lack of any ground truth assessments, I compare the accuracy of all methods as follows. For every method, I sample 100 pages each that it declares to be positives (with fidelity issues) and negatives (without fidelity issues). Two of this work's authors independently cross-checked each sample. We manually inspect the sampled archived pages to identify false positives and false negatives. For each positive, we check if the archived copy indeed has any missing content or broken functionality. For each negative, we spent at least a minute looking for differences between the archived copy and the original page. I found that the two checkers are largely in agreement in their assessments. The discrepancies are mainly due to minor differences between the live and archived pages that are difficult to notice (e.g., a small button missing at the bottom of the page).

False positives. As shown in Figure 5.8(a), across all four approaches, FIDEX has the lowest false positive rate. *More Errors* flags differences which do not impact the user, e.g., errors in unused libraries, tracking scripts, and third-party logins. Whereas, *Screenshot* and *Layout Tree* are unable to account for how dynamic components may be rendered differently in the loads of live and archived pages. FIDEX has the lowest false positive rate of 8–9%.

Not apparent from Figure 5.8(a) is that the time at which FIDEX records the layout tree



(a) (b)

Figure 5.8: FIDEX **versus three baselines, with respect to (a) false positives and (b) false negatives. The whiskers show the false positive/negative rate assessed by either checker.**

and associated JS writes is crucial. Like Browsertrix, if I record the page's state as soon as the browser's network activity ceases, then FIDEX's false positive rate increases significantly to 26–27%. FIDEX falls short of achieving zero false positives primarily because of timing. First, some web pages wait very long – up to a minute – before dynamism on the page kicks in. This is much longer than the timeout that FIDEX sets for each pending promise, to prevent itself from waiting forever. Second, on pages which keep adding elements forever (e.g., infinite scrolls), the layout tree captured by FIDEX can differ based on the time of capture.

False negatives. Since I capture screenshots after FIDEX waits to observe all dynamism, I observe that there are no false negatives with *Screenshot*. *More Errors* has more false negatives than FIDEX because many violations of fidelity do not result in any explicit errors. The 4 false negatives for FIDEX identified by both checkers are primarily caused by the incomplete tracking of layout information in my current implementation. Again, the timing of when FIDEX capture's a page's state matters; if I mimic Browsertrix's timing, FIDEX 's

false negative rate increases to 6—10% because some pages exercise the dynamic components which are broken only later.

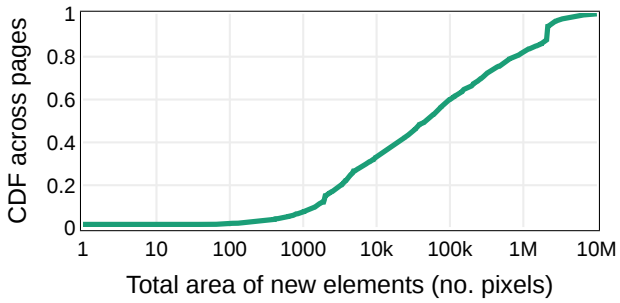
With *Screenshot* and *Layout Tree*, instead of looking for an exact match between the original and archived pages, I could check if the similarity is greater than a threshold. However, I have not found any good threshold for reducing false positives without a significant impact on false negatives. For example, for *Screenshot*, a similarity threshold of 99% offers the best F1-score and reduces the false positive rate by 10%. But, the false negative rate increases by 5%.

Impact of fidelity issues. Next, I examine the significance of the fidelity violations identified by FIDEX. I compute the area of the visual differences between the live and archived pages by collecting all leaf nodes that differ between the two layout trees. For each of these leaf nodes, I compute the relative location on the page normalized to the page's overall dimensions.

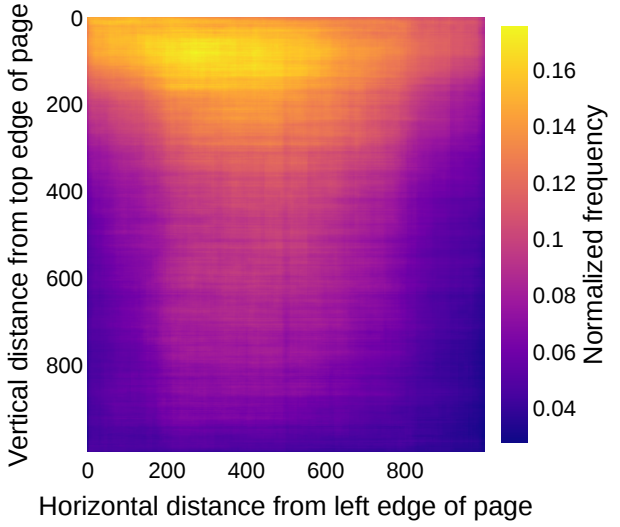
I find that, on the median archived page, the missing elements span a reasonably large area of 48K pixels. I also find that most of the content missing on the archived page appear near the top left corner of the original page. These results highlight the significance of the fidelity issues detected by FIDEX as users are more likely to perceive the impact if the area that differs is large and is closer to the top left of the page.

Figure 5.9(a) shows that the elements missing on the archived page often span a reasonably large area, with an area of 48K pixels on the median page. Pages with differences in the range 1K–10K pixels are typically missing small components such as buttons and icons. In the range 10K–100K, pages are usually missing an iframe (e.g., Trustpilot reviews) or a notification that spans the page width. Pages with differences of over 100K pixels are usually missing multiple components. For example, the main section of the page is missing, together with a change in the page's style. These are cases where errors that occur early in the page load preclude many subsequent executions that build the page.

Figure 5.9(b) shows that most of the affected components appear near the top left corner



(a)



(b)

Figure 5.9: **Impact of fidelity issues detected by** FIDEX**:** (a) Distribution of the area (in pixels) accounted for by the elements missing from the archived page. (b) A heat map of the locations on the page where the missing elements reside on the original page, normalized to a 1000x1000 square.

of the page. This highlights the significance of the fidelity issues detected by FIDEX as web developers place important content closer to the top and left margins of the page.

My manual examination of the pages on which fidelity is violated reveals that common visual fidelity issues include images that have been archived but are not rendered, incorrect styling of page content, and missing text. Functionally, the most common problems are navigation bars being rendered dysfunctional, and information that is supposed to pop up when an element is clicked does not.

5.4.3 Pinpointing of erroneous rewrites

Next, I use FIDEX to pinpoint the root cause for each fidelity violation that it identified. FIDEX pinpoints what it deems to be erroneous rewrites for 70%[4] of these archived pages. Roughly 30% of these are cases where the browser reports no additional errors when loading the archived page.

Validation of pinpointed errors. To validate how many of the erroneous rewrites pinpointed by FIDEX are indeed causes for the observed fidelity violations, I evaluate if the difference between the original page and its archived copy reduces either by selectively introducing the error in the former or eliminating it in the latter. For any erroneous rewrite that leads to a runtime error, I use pywb's proxy-mode to load the archived page without any rewriting but after injecting the error into the script where it occurred. For silent errors, FIDEX already validates them as part of its methodology for pinpointing erroneous rewrites (§5.2.2.2).

Since there are no existing methods to pinpoint erroneous rewriting in archival systems, I compare FIDEX with the baseline of *More Errors* described in the last section. This baseline simulates how developers would normally debug if the archived copy is problematic: by inspecting every additional error observed in the load of the archived copy.

I find that 92% of the erroneous rewrites pinpointed by FIDEX indeed lead to fidelity violations. In contrast, with *More Errors*, only 44% of the additional errors contribute to any fidelity issues. FIDEX greatly improves the accuracy of pinpointing by reducing the number of errors that developers need to investigate by more than 80%, while shining light on incorrect rewrites that do not result in any errors reported by the browser.

Clustering of pinpointed errors. Although FIDEX is able to pinpoint root causes for thousands of archived copies that violate fidelity, it cannot automatically fix erroneous rewrites. Since it is impractical for developers to look at FIDEX's output for thousands

[4]I expect this percentage to be less than 100 because, as observed previously, some of the fidelity violations flagged by FIDEX are false positives.

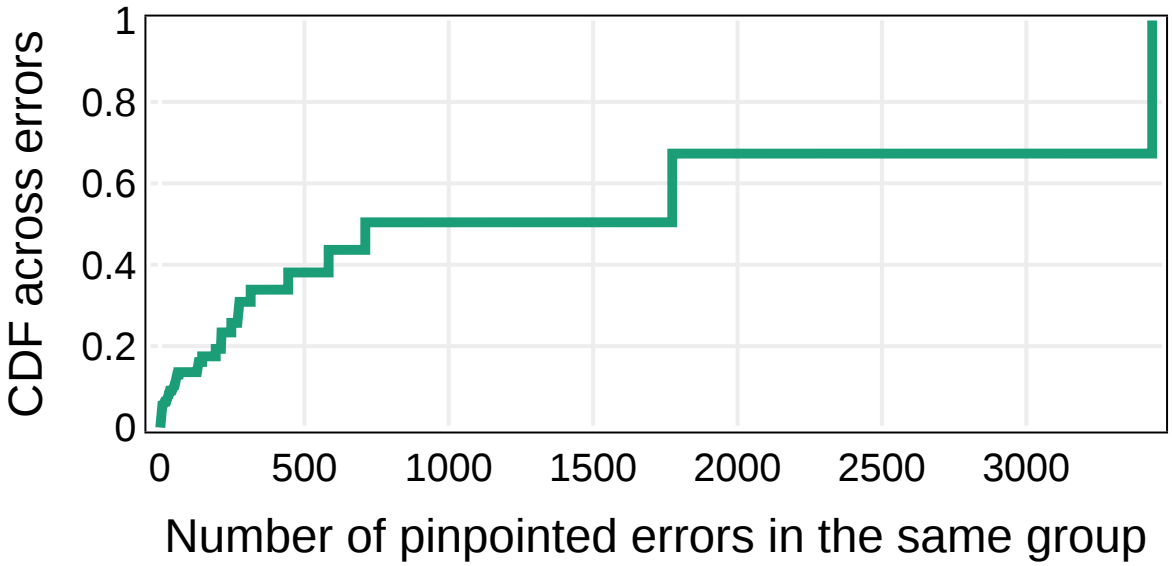


Figure 5.10: **For each erroneous rewrite that FidEx pinpoints, number of other similar pinpointed errors.**

of pages, I group pinpointed errors that result from the same rewriting mechanism. For this, I leverage the error messages associated with runtime errors. I follow JavaScript error reference [6] to get a list of error message templates, and group errors that follow the same template but differ only in the page-specific variable. For silent errors, since FidEx adds specification checking to overridden methods, it logs a a message which describes the observed problem. I simply cluster such errors based on my customized error messages.

Figure 5.10 shows a heavy-tailed distribution: the 4 largest error groups account for more than 60% of the 10,509 errors pinpointed by FidEx. On the other hand, I find that there is a long tail of esoteric issues. For example, 295 groups correspond to erroneous rewrites seen on no more than 5 pages. As the web and systems for archiving it evolve, I expect that some of these issues may become more prevalent.

5.4.4 Fixing commonly pinpointed errors

Over the past several months, I worked with the developers of pywb to fix the bugs in pywb which result in the 4 most common errors pinpointed by FidEx as the root cause for fidelity

violations.

- **Syntax errors:** The most frequent source of fidelity violations are syntax errors triggered by invalid keywords (e.g., `arguments`) in strict mode. Since ECMAScript modules (ESM) [55] operate in strict mode by default, pywb's inability to detect them led to incorrect rewrites. I fixed the problem by detecting ESM and applying a separate rewrite method that avoids syntax errors while preserving the intended rewriting effects.
- **Window parent/top:** The second most common issue involves silent errors where `Window.top` and `Window.parent` fail to return the correct parent frame. Under frameless mode (used by the Wayback Machine), the check for whether the current frame is the top frame always returns true, causing incorrect `top/parent` values. I resolve this by correcting the frame-check logic.
- **Cross-origin frame access:** The third category of common errors arises when rewritten attribute accesses are blocked (silently) in cross-origin frames, such as `srcdoc`. I fix this by removing unnecessary cross-frame attribute accesses in rewrites and relaxing the safety policy to allow legitimate accesses.
- **Reference errors:** Lastly, I address reference errors caused by scoping of global variables after rewrites. I parse each JavaScript file into an Abstract Syntax Tree (AST) and replace all global variable declarations with `var`, ensuring they remain in the global scope.

For each of the 4 target errors, I evaluate the effectiveness of my fix on 400 archived pages affected by that error. As I see in Figure 5.11, all 4 fixes ensure that pywb is now able to serve more than 60% of the previously affected archived copies without any loss in fidelity. For an additional 11–39%, my fixes resolve the target errors but some new errors prevent full recovery. In 0–12% of cases, my fixes fail to resolve the target errors, which I attribute to factors such as incomplete fixes (e.g., AST parsing failures), undetected silent errors, or inaccuracies in FIDEX's diagnosis.

I also evaluated the impact of my modifications to pywb on 400 archived copies which it was already able to serve without any issues. FIDEX flagged fidelity violations in 3 of these

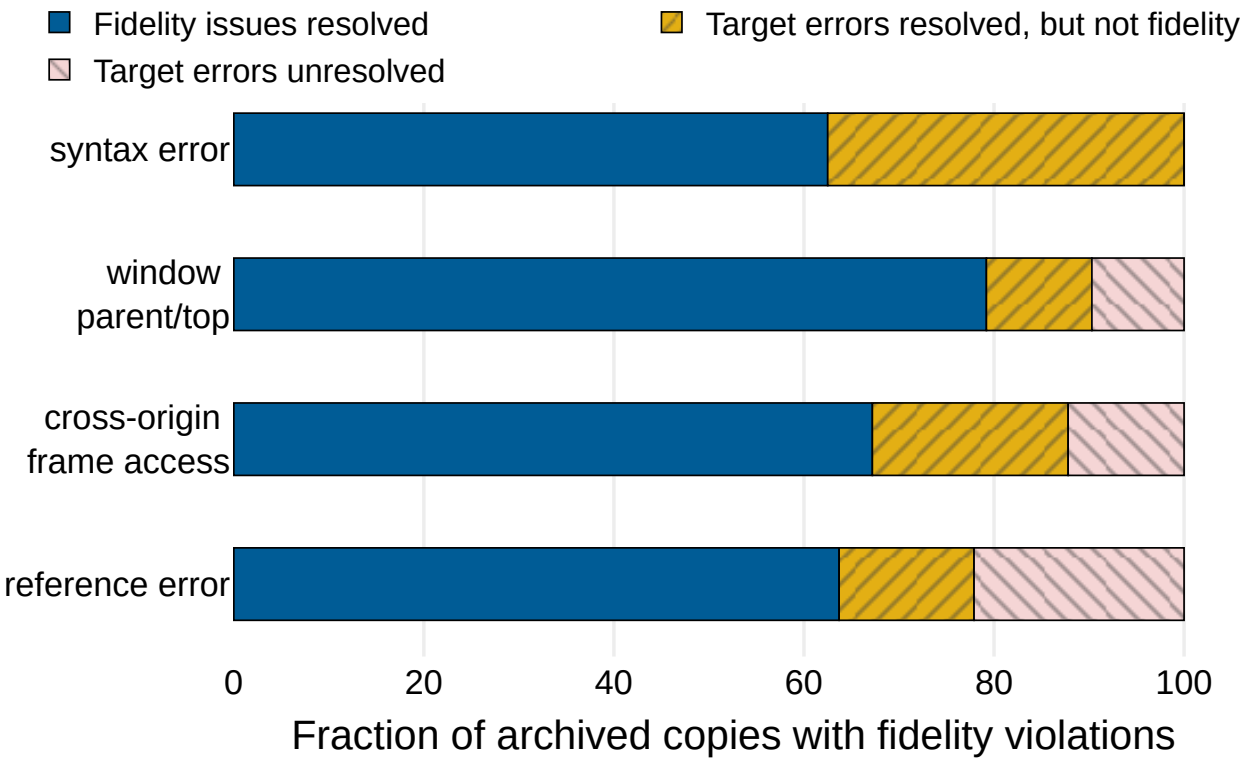


Figure 5.11: **Breakdown of the impact of my modifications to pywb on 400 low-fidelity archived copies.**

400 pages. However, manual inspection revealed that all of these were false positives.

Finally, I evaluated my new version of pywb on my full dataset of 80K pages. The fraction of archived pages on which FIDEX identified a fidelity violation reduced from 15% to 9%. For example, both the problematic cases from Figure 2.4 are now resolved.

5.4.5 Efficiency

I evaluate FIDEX's efficiency from three perspectives: crawling overhead, runtime for fidelity violation detection, and runtime for pinpointing.

Crawling overhead. I run crawlers on a server with a 32-core 2.1 GHz Intel Xeon CPU, a 1 Gbps network connection, and 128 GB memory. I launch 16 crawlers in parallel to saturate all cores. I compare the baseline crawler with no instrumentation against FIDEX's crawler. I also perform this comparison after augmenting both crawlers to capture a screenshot after

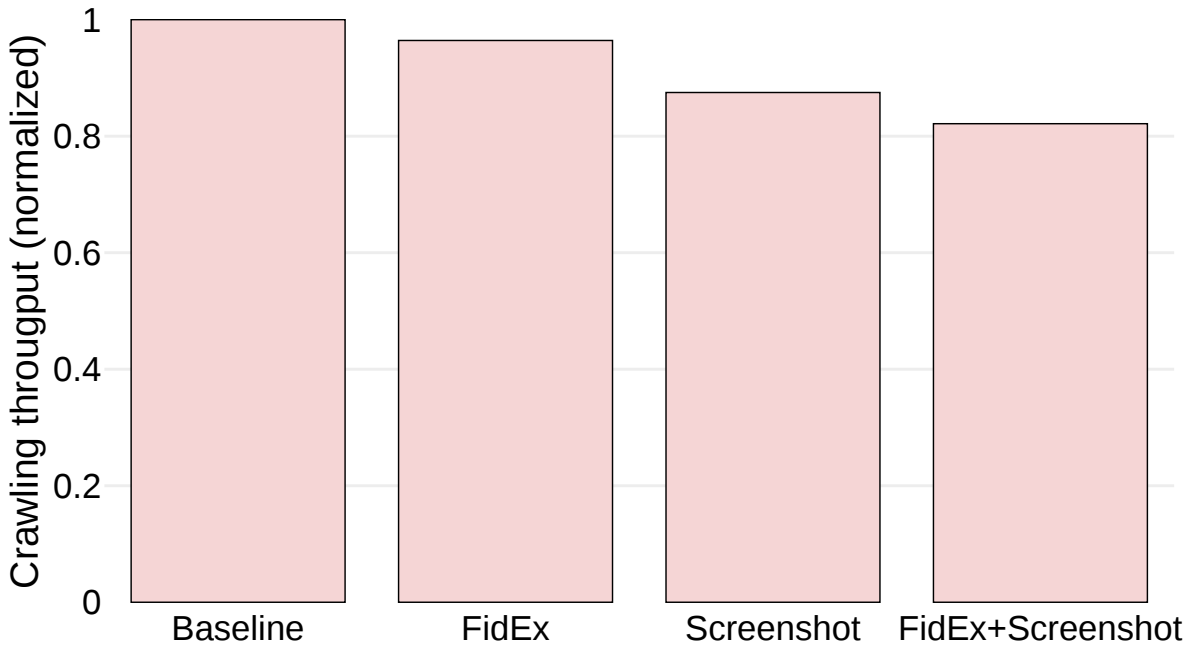
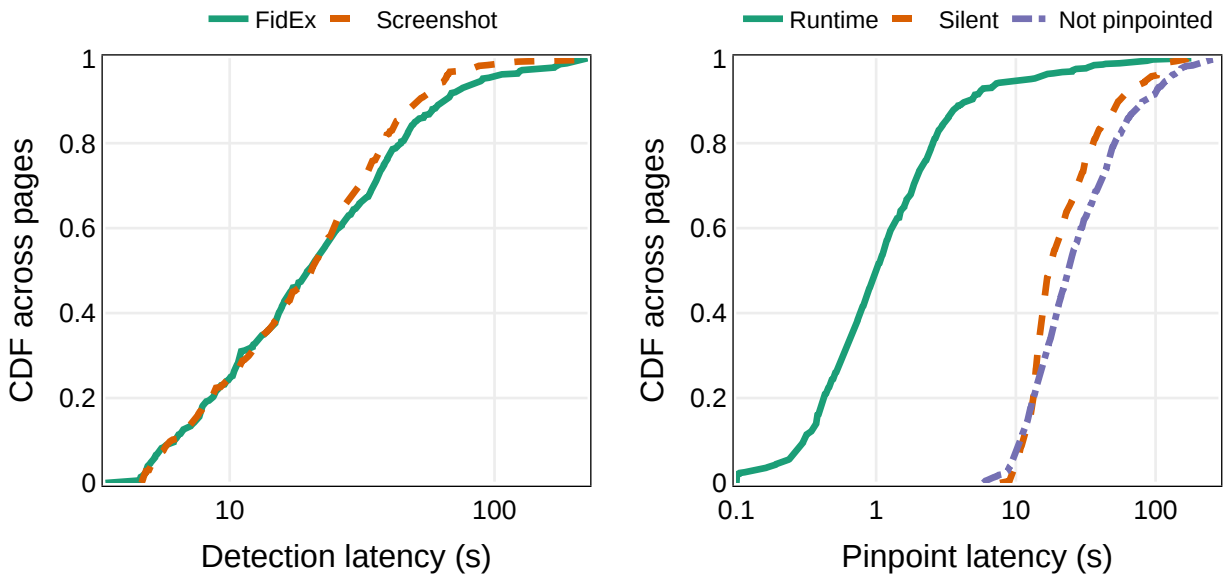


Figure 5.12: **Comparison of crawling throughput, normalized to that offered by baseline crawler without any instrumentation.**

the initial load and after every interaction. Capturing a screenshot when running FIDEX can help users understand the fidelity violations identified by it. Figure 5.12 shows the normalized crawling throughput across these crawling strategies. Compared to a baseline crawler, FIDEX's tracking of JavaScript writes to the DOM and CSSOM reduces crawling throughput by 5%. However, FIDEX still incurs lower overhead compared to taking screenshots, which degrades throughput by 13% compared to the baseline.

Runtime for fidelity violation detection. After FIDEX crawls a page and loads its archived copy, I measure how long it takes to process the information captured and compute the difference between the two layout trees. Figure 5.13(a) shows that, for the median page, FIDEX takes a median of 19.8 seconds to determine whether it violates fidelity. This latency, which includes the time to load the archived copy, is similar to the median of 20.4 seconds with screenshot-based detection. Figure 5.13(a) also shows that, for roughly 60% of a random sample of 1,000 archived pages, FIDEX's detection latency is comparable to – or even lower than – that of the screenshot-based method. This is because the latency is largely dominated



(a) (b)

Figure 5.13: FIDEX**'s runtime to (a) evaluate each archived copy's fidelity, and (b) pinpoint the root cause on every page deemed to violate fidelity. The latter separately shows when pinpointing is based on runtime errors, pinpointing is based on silent errors, or** FIDEX **is unable to pinpoint any erroneous rewrites.**

by the time needed to load an archived page and capture relevant state, for which FIDEX has lower overhead. FIDEX takes longer to process larger pages which produce bigger layout trees and more JS writes. I expect this overhead can be reduced, as my implementation of FIDEX's comparison of layout trees is not yet optimized, whereas I compare screenshots using mature, highly-optimized libraries.

Runtime for pinpointing. On a random sample of 1,000 pages flagged by FIDEX as violating fidelity, Figure 5.13(b) shows the distribution of latency for FIDEX to pinpoint pywb's problematic rewrites. FIDEX takes a median latency of only 1 second to identify which of the runtime errors in the load of an archived page are responsible for observed fidelity violations. For silent errors, FIDEX takes a median of 17 seconds to pinpoint the erroneous rewrites if it is able to do so, and 24 seconds if it fails to pinpoint anything. This is because FIDEX needs to reload the archived copy after selectively removing each silent error in order to validate its impact. When FIDEX fails to pinpoint any errors, it requires

24 seconds on the median page.

5.4.6 Generalizability

Lastly, I evaluate the broader applicability of FidEx beyond fidelity issues introduced by pywb on pages archived from sites in the Tranco list.

Alternate datasets. I run FidEx on pages derived from two datasets whose preservation is important.

- The End of Term (EOT) dataset [121] captures a snapshot of every publicly available federal website once every four years. The intent is to document changes in the web presence of the Executive Branch of the US government at the time of presidential transition.
- The Collaborative ART Archive (CARTA) [1] dataset includes at-risk web-based art materials captured collaboratively by the Internet Archive and the New York Arts Resources Consortium (NYARC). Pages in this dataset have rich visual and interactive content.

From either dataset, I randomly sample 2,000 pages that exist on the web today and pass my filtration criteria (§5.4.1).

FidEx finds fidelity violations on 12% of the pages from EOT and 11% of the pages from CARTA. For pages with fidelity violations, FidEx is able to pinpoint root causes on 81% and 72% of the pages, respectively, from EOT and CARTA. These numbers are close to the corresponding fractions seen in my dataset from Tranco. I find that fidelity issues from EOT and CARTA include both missing content (e.g., on the archived copy of the Smithsonian National Postal Museum's blog [12], images for individual blog posts fail to load) and broken functionality (e.g., on NASA's page about climate change [4], switching from the default tab about 'Evidence' to the tabs about 'Causes' and 'Effects' does not work).

Alternate archival system. To understand whether other archival systems too suffer from fidelity issues, I use FidEx to examine pages archived with ReplayWeb.page [18], a serverless archival system which runs directly in the browser. I made a number of modifications to my implementation of FidEx in order to account for the differences between

ReplayWeb.page and pywb. These differences include ReplayWeb.page's reliance on service workers to serve resources, its loading of any archived page within an iframe, and different rewriting outcomes on JavaScript.

I use FidEx to evaluate the fidelity with which ReplayWeb.page serves 2,000 archived pages. FidEx finds evidence of fidelity violations on 10% of these pages. 42% of the pages served with poor fidelity by ReplayWeb.page were also flagged by FidEx when these pages were loaded using pywb, but 56% of the problems were unique to ReplayWeb.page. This result implies that ReplayWeb.page has many bugs that are not in pywb.

Because my implementation of FidEx that is compatible with ReplayWeb.page is not yet as mature as my implementation for pywb, FidEx is able to pinpoint erroneous rewrites for only 40% of the archived pages with poor fidelity.

5.5 Summary

Web archives devote a significant amount of person hours and infrastructure to crawl and preserve pages from the web. It is a shame, then, that the widespread use of JavaScript on the web hampers their ability to serve archived pages with high fidelity. In this work, I showed how to more reliably identify fidelity violations than feasible with existing methods and help inform the diagnosis of the bugs in archival systems that cause these problems. The key was to track JavaScript execution and associate it with the appearance and functionality of a page. Building on my collaboration with the developers of pywb, I plan to work with several web archive practitioners to incorporate FidEx's methods into their workflow.

CHAPTER 6

Conclusion and Future work

The World Wide Web (WWW) has grown tremendously since its inception. Yet this growth has also led to an increasing number of links becoming inaccessible over time—-a phenomenon known as *link rot*. Today, web archives are often used to cope with the link rot, by capturing snapshots of live web pages before they disappear. However, the evolution of modern web pages introduces significant new challenges for combating link rot.

First, the sheer scale of today's web makes it infeasible for archives to capture every page each time it changes. Consequently, even in large archives such as the Internet Archive's Wayback Machine, many pages either lack an archived copy altogether or only have stale versions that lag far behind the live site.

Second, the increasing complexity of modern web pages has made it much harder to preserve their fidelity. With widespread reliance on JavaScript, archives must 1) employ browser-based crawlers to capture all resources needed for rendering, and 2) accurately rewrite JavaScript APIs so that AJAX requests are redirected to the archive rather than the original host. Both tasks demand substantial crawling resources and engineering effort, which most archives cannot fully sustain. As a result, many archived pages suffer from missing resources or erroneous script rewrites, leading to degraded appearance and functionality compared to their live-web counterparts.

6.1 Thesis Contributions

In this thesis, I make the following contributions to solve link rot on the modern web:

A system to revive dead links when web archives fail to patch live web: Many broken links do not stem from their pages' deletion but rather from pages being reorganized to new URLs (aliases). My first contribution, FABLE, automatically discover aliases for broken links. The key insights behind FABLE are that pages on a site are rarely reorganized in isolation; instead, they are often moved together following site-wide structural changes. FABLE learns and exploits these reorganization patterns to accurately and efficiently find new aliases of broken links. FABLE increases the fraction of dead links for which the corresponding new URLs can be found by 50%, while reducing the median delay incurred in identifying the new URL for a broken link from over 40 seconds to less than 10 seconds.

A study to optimize efficiency–fidelity tradeoffs for archival crawlers: Modern web pages rely heavily on AJAX fetches to render, which require executing JavaScript during crawling. While browser-based crawlers can capture these resources, they are far more expensive than static crawlers and cannot be applied universally at web scale. My second contribution is a conducted study towards better tradeoffs between archived copies' fidelity and crawling efficiency. I show that by strategically selecting a small subset of pages to crawl dynamically, web archives can preserve fidelity for the vast majority of all crawled pages. By combining a set of techniques, I find that by dynamically crawling only 8.5% of pages, 98% of the remaining static crawls avoid fidelity violations.

A system to detect and diagnose errors in serving archived copies: Even when all necessary resources are archived, web archives must rewrite resources so that dependent fetches–especially JavaScript initiated AJAX calls–are redirected to the archive's host. This process requires comprehensive modifications to JavaScript APIs and is often incorrect, leading to fidelity violations in archived copies. My last contribution, FIDEX, automatically detect fidelity violations caused by faulty archive edits and diagnose their root causes to

guide developers in fixing them. By carefully tracking and comparing JavaScript execution between loads from the live web and archive, FidEx accurately identifies fidelity issues and pinpoints them to a large number of different root causes. In evaluation, FidEx reduces false positive rate of detection from 70% to 10%, and after fixing the most commonly pinpointed errors from FidEx, I reduced the fraction of archived pages which violate fidelity from 15% to 9%.

6.2 Long-term applicability

The contributions presented in this thesis provided concrete solutions to link rot on today's web and address key challenges in existing approaches. However, the web is constantly evolving. Many of the specific techniques introduced here may eventually become obsolete. For instance, aliases discovered by Fable can themselves break over time; the efficiency–fidelity tradeoff for archival crawling that I demonstrated may shift as web pages become even more JavaScript-heavy; and new Web APIs introduced in browsers could lead to new classes of fidelity violations that FidEx may not detect or diagnose.

Nevertheless, I believe the lessons and insights from this thesis have long-term applicability. They offer general principles that can guide future developers and researchers in combating link rot and improving the resilience of the web archival ecosystem.

Broken links ≠ deleted pages. When users encounter a link that no longer works, they often assume the corresponding page has been removed from the web. My work has shown that this assumption is frequently incorrect. Many "broken" links result from site reorganizations, where pages remain available but under new URLs (aliases). As long as site administrators continue to restructure their sites, such aliases will persist. Moreover, as site reorganization tools become more common, the underlying transformation patterns connecting old and new URLs will become even more detectable. Thus, systems like Fable, which learn and exploit these patterns, will remain valuable for reviving broken links.

Resources are shared across pages. Web pages will continue to grow more dynamic, relying increasingly on JavaScript execution and client-initiated fetches during load. While this trend makes dynamic crawling more important for preserving fidelity, the fundamental observation that resources are widely shared across pages and over time will remain true. No matter how web technologies evolve, developers will continue to build sites using public components, widgets, and frameworks. As a result, many of the resources needed for rendering a page will already exist in other pages under the same site. Leveraging this redundancy will always be a powerful way to achieve efficient yet high-fidelity archival crawling.

The critical render path defines fidelity. FIDEX's accurate detection of fidelity violations builds upon representations extracted from a browser's critical render path–the core sequence of operations determining how a page is visually and functionally realized. In the future, browsers may introduce new mechanisms to render a page. However, as long as a page's rendering process largely remains deterministic given a fixed set of resources, analyzing and comparing critical render paths will continue to be essential for determining whether two page loads—archived or live—are truly equivalent.

6.3 Future Work

The modern web will continue to evolve, with new pages being created, links breaking over time, and page loading becoming increasingly complex. I believe the contributions presented in this thesis to address link rot will remain broadly compatible with these future developments. At the same time, the emergence of Large Language Models (LLMs) opens new opportunities to not only mitigate link rot but also enable novel ways of interacting with the web. In this section, I outline potential directions for future research inspired by this thesis.

Reviving dead links with LLMs. In FABLE, alias candidates are currently discovered either by querying search engines or by inferring patterns through programming-by-example

(PBE). Both techniques show promising potential when combined with Large Language Models (LLMs) [104, 134]. LLMs can improve query generation from archived copies, enhance information retrieval, and strengthen pattern learning and inference. These improvements can increase the likelihood of discovering and verifying aliases, thereby reviving a larger fraction of dead links.

Furthermore, page reorganization is not always a one-to-one mapping: multiple pages may be merged into a single page, or a single page may be split into several. Such cases remain beyond FABLE's current capabilities. However, both the context of links and the content of the target pages, link revival could extend beyond the page level to recover content within or across reorganized pages.

A clean-slate design of web archives. Most existing web archives load archived copies through a browser in the same way that live web pages are loaded. This design forces archives to rewrite pages so that dependent resource requests are redirected to the archive's host. While some archives avoid this complexity by serving static artifacts such as screenshots [38, 61], these approaches often fail to preserve the user's navigation experience, including interactions and dynamic components.

However, web archives need not replicate the exact loading process of the live web, where the initial HTML is progressively modified by executing JavaScript. In the live web, this design is crucial to decouple components–often from different parties–so that each can be updated independently. By contrast, archived copies are fully self-contained, and their rendering process is fixed at the time of capture. This opens the door to a clean-slate design: instead of serving pages piece by piece, archives could compile, redesign, or even generate fully self-contained representations—potentially leveraging LLMs—to deliver the entire page at once. Such an approach could simplify replay, reduce computational complexity for both archives and users, and preserve the essence of the archived experience.

Rethinking the design of web links. The web was originally inspired by Vannevar Bush's vision in *As we may think* [70]. Hyperlinks were conceived as "associative trails,"

connecting information in ways that mirrored human thought. However, when a link becomes inaccessible (link rot) or when the target page undergoes substantial changes (content drift), the link no longer serves its intended purpose.

A central reason this problem persists is that links today are unidirectional: any page can point to another independently, without coordination. Tim Berners-Lee once proposed an alternative design in which links would be bidirectional, with both sides aware of the connection [67]. While this was never realized, I believe new designs for web links could fundamentally mitigate link rot and content drift while remaining compatible with today's web architecture. For example:

- **Archival on link creation:** When a link is created, the linking site could capture and store an archived copy of the target, if the intent is to reference specific content. Yet, links today often point not just to static content but to functionality (e.g., search, chat, live streaming). Understanding the intent behind a link, and designing mechanisms to preserve it, remains an open challenge.
- **Versioning:** Similar to version control systems in software, pages could be versioned so that links point to a specific snapshot, thereby preventing content drift. However, defining what constitutes a "version" for modern dynamic pages, and doing so with minimal serving overhead, requires further study.
- **Protocols for reorganization:** When pages are reorganized, linking sites could be notified or able to query for updated URLs. Designing such protocols in a way that balances efficiency and utility poses an important research challenge.

I have proposed preliminary work in this space [84], but more comprehensive research is needed to explore how such mechanisms could prevent link rot at its source.

Archiving and crawling in the era of LLMs. Traditionally, web crawlers have primarily served information retrieval, while archives have focused on enabling users to navigate past pages with high fidelity. Today, however, LLMs are deeply intertwined with the web: from pre-training that relies on large-scale public web crawl datasets such as Common Crawl [49],

to emerging web agents [79, 141] that continuously interact with online content.

This shift raises new research questions for the design of crawlers and archives. For example: 1) What data should be crawled—and how should it be captured—to improve downstream model quality after pre-training? 2) How can archived copies be incorporated into retrieval-augmented generation to provide a time-aware perspective on knowledge? 3) What new interaction mechanisms could web agents use, beyond the traditional browser interfaces originally designed for human users?

6.4 Summary

In this thesis, I presented three solutions to address link rot on the modern web: FABLE, FIDEX, and a study on better fidelity–efficiency tradeoffs in web archives. Each work addresses key limitations in current existing approaches. I hope this dissertations can serve as a guide for participants in the web ecosystem– such as web archives and site hosts—by offering more effective methods to preserve access when links inevitably cease to work.

BIBLIOGRAPHY

[1] About the collaborative art archive (CARTA). https://carta.archive-it.org/.

[2] beautifulsoup4 · pypi. https://pypi.org/project/beautifulsoup4/.

[3] Blog | national postal museum. https://web.archive.org/web/20250726012058/https://postalmuseum.si.edu/blog.

[4] Climate change - NASA science. https://science.nasa.gov/climate-change/.

[5] Esprima. https://esprima.org/.

[6] Javascript error reference - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors.

[7] Library of congress. https://www.loc.gov/web-archives/.

[8] Playback - IIPC. https://netpreserve.org/web-archiving/playback/.

[9] Puppeteer. https://pptr.dev/.

[10] Render-tree construction, layout, and paint | articles | web.dev. https://web.dev/articles/critical-rendering-path/render-tree-construction.

[11] Replay fails (looks like a rewriting issue): Error setting href = index.cfm?tem=2&spr=0&hpn=1&idn=999999 · issue #789 · webrecorder/pywb. https://github.com/webrecorder/pywb/issues/789.

[12] Smithsonian National Postal Museum. https://postalmuseum.si.edu/blog.

[13] The half-life of a link is two year — blog.zomdir.com. https://blog.zomdir.com/2017/10/the-half-life-of-link-is-two-year.html.

[14] UK Web Archive currently unavailable — webarchive.org.uk. https://www.webarchive.org.uk/.

[15] The WARC format 1.1. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.1/.

[16] Web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API.

[17] webrecorder/pywb: Core python web archiving toolkit for replay and recording of web archives. https://github.com/webrecorder/pywb.

[18] webrecorder/replayweb.page: Serverless replay of web archives directly in the browser. https://github.com/webrecorder/replayweb.page.

[19] webrecorder/wombat: Wombat.js client-side rewriting library. https://github.com/webrecorder/wombat.

[20] KDE 1.92 release announcement. https://web.archive.org/web/20060209082707/http://www.kde.org:80/announcements/announce-1.92.html, 2006.

[21] What if? (2008) #1 | comic books | comics | marvel.com. http://web.archive.org/web/20121017122005/http://marvel.com/comic_books/issue/22962/what_if_2008_1, 2012.

[22] After the revolution: Youth, democracy, and the politics of disappointment in Serbia - Jessica Greenberg. http://web.archive.org/web/20140701030455/http://sup.org/book.cgi?id=21682, 2014.

[23] Harvard kennedy school - mossavar-rahmani center for business and government :: About :: Fellows :: Senior fellows: 2017-2018 (copy on July 12, 2017). https://web.archive.org/web/20170712144006/http://www.hks.harvard.edu/centers/mrcbg/about/fellows/currentsrfellows, 2017.

[24] Brave browser and the Wayback Machine: Working together to help make the web more useful and reliable. http://blog.archive.org/2020/02/25/brave-browser-and-the-wayback-machine-working-together-to-help-make-the-web-more-useful-and-reliable/, 2020.

[25] Cloudflare and the Wayback Machine, joining forces for a more reliable web. https://blog.archive.org/2020/09/17/internet-archive-partners-with-cloudflare-to-help-make-the-web-more-useful-and-reliable/, 2020.

[26] 410 gone - HTTP. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/410, 2023.

[27] Alexa - competitive analysis, marketing mix, and website traffic. https://www.alexa.com/siteinfo, 2023.

[28] Canonical link element - wikipedia. https://en.wikipedia.org/wiki/Canonical_link_element, 2023.

[29] Category:articles with dead external links - wikipedia. https://en.wikipedia.org/wiki/Category:Articles_with_dead_external_links, 2023.

[30] Category:articles with permanently dead external links - wikipedia. https://en.wikipedia.org/wiki/Category:Articles_with_permanently_dead_external_links, 2023.

[31] chromium/dom-distiller: Distills the DOM. https://github.com/chromium/dom-distiller, 2023.

[32] Internet Archive: Wayback Machine. https://archive.org/web/, 2023.

[33] InternetArchiveBot. https://meta.wikimedia.org/wiki/InternetArchiveBot, 2023.

[34] Ipfs powers the distributed web. https://ipfs.tech/, 2023.

[35] Klazify - free website categorization & logo api. find company's category and logo from url. https://www.klazify.com/, 2023.

[36] Medium sitemap. https://medium.com/sitemap/sitemap.xml, 2023.

[37] Newspaper3k: Article scraping & curation — newspaper 0.0.2 documentation. https://newspaper.readthedocs.io/en/latest/, 2023.

[38] Perma.cc. https://perma.cc/, 2023.

[39] Prose - text transformation - microsoft research. https://www.microsoft.com/en-us/research/project/prose-text-transformation/usage/, 2023.

[40] Public suffix list. https://publicsuffix.org/, 2023.

[41] Robust links - make your link robust. https://robustlinks.mementoweb.org/, 2023.

[42] Stack exchange data dump : Stack exchange, inc. : Free download, borrow, and streaming : Internet archive. https://archive.org/details/stackexchange, 2023.

[43] User:fablebot/new urls for permanently dead external links - wikipedia. https://en.wikipedia.org/wiki/User:FABLEBot/New_URLs_for_permanently_dead_external_links, 2023.

[44] Using Flash Fill in Excel. https://support.microsoft.com/en-us/office/using-flash-fill-in-excel-3f9bcf1e-db93-4890-94a0-1578341f73f7, 2023.

[45] Amazon Go — amazon.jobs. https://www.amazon.jobs/content/en/teams/worldwide-grocery-stores/amazon-go, 2025.

[46] Bolt Design System — boltdesignsystem.com. https://boltdesignsystem.com/pattern-lab/patterns/40-components-code-snippet-10-code-snippet-language/40-components-code-snippet-10-code-snippet-language.html/, 2025.

[47] Bolt Design System — web.archive.org. https://web.archive.org/web/20250218130712/https://boltdesignsystem.com/pattern-lab/patterns/40-components-code-snippet-10-code-snippet-language/40-components-code-snippet-10-code-snippet-language.html/, 2025.

[48] Browsertrix crawler. https://github.com/webrecorder/browsertrix-crawler, 2025.

[49] Common crawl - open repository of web crawl data. https://commoncrawl.org/, 2025.

[50] Critical rendering path - performance | mdn. https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/Critical_rendering_path, 2025.

[51] GitHub - ArchiveBox/ArchiveBox: Open source self-hosted web archiving. https://github.com/ArchiveBox/ArchiveBox, 2025.

[52] GitHub - internetarchive/brozzler: brozzler - distributed browser-based web crawler — github.com. https://github.com/internetarchive/brozzler, 2025.

[53] GitHub - internetarchive/heritrix3: Heritrix is the Internet Archive's open-source, extensible, web-scale, archival-quality web crawler project. — github.com. https://github.com/internetarchive/heritrix3, 2025.

[54] Japanese Kit Kat: Chocolate Orange — bokksumarket.com. https://www.bokksumarket.com/collections/best-seller/products/japanese-kit-kat-choco-orange, 2025.

[55] Javascript modules - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules, 2025.

[56] Next.js by Vercel - The React Framework — nextjs.org. https://nextjs.org/, 2025.

[57] pywb/pywb/rules.yaml at main · webrecorder/pywb — github.com. https://github.com/webrecorder/pywb/blob/main/pywb/rules.yaml, 2025.

[58] Review crawl quality - Browsertrix docs. https://docs.browsertrix.com/user-guide/review/, 2025.

[59] third_party/blink/renderer/core/layout/layout_object.h - chromium/src - git at google. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/third_party/blink/renderer/core/layout/layout_object.h, 2025.

[60] Usage — pywb 2.7 documentation. https://pywb.readthedocs.io/en/latest/manual/usage.html#http-s-proxy-mode-access, 2025.

[61] Webpage archive. https://archive.ph/, 2025.

[62] Webrecorder. https://webrecorder.net/, 2025.

[63] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[64] Scott G Ainsworth, Ahmed Alsum, Hany SalahEldeen, Michele C Weigle, and Michael L Nelson. How much of the web is archived? In *ACM/IEEE Joint Conference on Digital Libraries*, 2011.

[65] Ahmed AlSum, Michele C Weigle, Michael L Nelson, and Herbert Van de Sompel. Profiling web archive coverage for top-level domain and content language. *International Journal on Digital Libraries*, 2014.

[66] Ziv Bar-Yossef, Andrei Z Broder, Ravi Kumar, and Andrew Tomkins. Sic transit gloria telae: Towards an understanding of the web's decay. In *WWW*, 2004.

[67] Timothy J Berners-Lee. Information management: A proposal. Technical report, 1989.

[68] Andrei Broder. A taxonomy of web search. In *ACM SIGIR Forum*, 2002.

[69] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, page 473–484, New York, NY, USA, 2013. Association for Computing Machinery.

[70] Vannevar Bush et al. As we may think. *The atlantic monthly*, 176(1):101–108, 1945.

[71] G. Candea and S. Andrica. Warr: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, pages 403–410, Los Alamitos, CA, USA, jun 2011. IEEE Computer Society.

[72] Soumen Chakrabarti, Kunal Punera, and Mallela Subramanyam. Accelerated focused crawling through online relevance feedback. In *Proceedings of the 11th international conference on World Wide Web*, pages 148–159, 2002.

[73] Soumen Chakrabarti, Martin Van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific web resource discovery. *Computer networks*, 31(11-16):1623–1640, 1999.

[74] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. Technical report, Stanford, 1999.

[75] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. In *Proceedings of the 11th international conference on World Wide Web*, pages 124–135, 2002.

[76] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 171–180, 2012.

[77] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. X-pert: Accurate identification of cross-browser issues in web applications. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 702–711. IEEE, 2013.

[78] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. WebDiff: Automated identification of cross-browser issues in web applications. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[79] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

[80] Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, C Lee Giles, Marco Gori, et al. Focused crawling using context graphs. In *VLDB*, pages 527–534. Citeseer, 2000.

[81] Dennis Fetterly, Mark Manasse, and Marc Najork. On the evolution of clusters of near-duplicate web pages. In *IEEE/LEOS International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices*, 2003.

[82] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet L Wiener. A large-scale study of the evolution of web pages. *Software: Practice and Experience*, 34(2):213–237, 2004.

[83] Ayush Goel, Jingyuan Zhu, and Harsha V. Madhyastha. Making links on your web pages last longer than you. In *HotNets*, 2022.

[84] Ayush Goel, Jingyuan Zhu, and Harsha V Madhyastha. Making links on your web pages last longer than you. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 145–151, 2022.

[85] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V. Madhyastha. Jawa: Web archival in the era of JavaScript. In *OSDI*, 2022.

[86] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V Madhyastha. Jawa: Web archival in the era of JavaScript. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 805–820, 2022.

[87] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V. Madhyastha. Sprinter: Speeding up High-Fidelity crawling of the modern web. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 893–906, Santa Clara, CA, April 2024. USENIX Association.

[88] Daniel Conrad Halbert. *Programming by example*. PhD thesis, University of California, Berkeley, 1984.

[89] Terry L Harrison and Michael L Nelson. Just-in-time recovery of missing web pages. In *ACM Conference on Hypertext and Hypermedia*, 2006.

[90] Monika Henzinger. Finding near-duplicate web pages: A large-scale evaluation of algorithms. In *SIGIR*, 2006.

[91] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[92] Zhongjun Jin, Michael R Anderson, Michael Cafarella, and HV Jagadish. Foofah: Transforming data by example. In *SIGMOD*, 2017.

[93] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[94] Shawn M Jones, Herbert Van de Sompel, Harihar Shankar, Martin Klein, Richard Tobin, and Claire Grover. Scholarly context adrift: Three out of four URI references lead to changed content. *PloS one*, 2016.

[95] Terence Kelly and Jeffrey Mogul. Aliasing on the world wide web: Prevalence and performance implications. In *Proceedings of the 11th international conference on World Wide Web*, pages 281–292, 2002.

[96] Martin Klein and Michael L Nelson. Revisiting lexical signatures to (re-) discover web pages. In *International Conference on Theory and Practice of Digital Libraries*, pages 371–382. Springer, 2008.

[97] Martin Klein and Michael L Nelson. Evaluating methods to rediscover missing web pages from the web infrastructure. In *ACM/IEEE Joint Conference on Digital Libraries*, 2010.

[98] Martin Klein, Jeffery Shipman, and Michael L Nelson. Is this a good title? In *ACM Conference on Hypertext and Hypermedia*, 2010.

[99] Martin Klein, Jeb Ware, and Michael L Nelson. Rediscovering missing web pages using link neighborhood lexical signatures. In *ACM/IEEE Joint Conference on Digital Libraries*, 2011.

[100] Wallace Koehler. Web page change and persistence—a four-year longitudinal study. *Journal of the American society for information science and technology*, 53(2):162–171, 2002.

[101] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. Boilerplate detection using shallow text features. In *WSDM*, 2010.

[102] John Kunze and Richard Rodgers. The ark identifier scheme. 2008.

[103] Steve Lawrence, Frans Coetzee, Eric Glover, Gary Flake, David Pennock, Bob Krovetz, Finn Nielsen, Andries Kruger, and Lee Giles. Persistence of information on the web: Analyzing citations contained in research articles. In *CIKM*, 2000.

[104] Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *Advances in Neural Information Processing Systems*, 37:44761–44790, 2024.

[105] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019.

[106] Yi Li, Shaohua Wang, and Tien Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661–673. IEEE, 2021.

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

[108] Sonal Mahajan and William G. J. Halfond. Detection and localization of html presentation failures using computer vision-based techniques. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[109] Sonal Mahajan and William G.J. Halfond. Finding html presentation failures using image comparison techniques. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 91–96, New York, NY, USA, 2014. Association for Computing Machinery.

[110] Sonal Mahajan and William GJ Halfond. Websee: A tool for debugging html presentation failures. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8. IEEE, 2015.

[111] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[112] James Mickens. Silo: Exploiting {JavaScript} and {DOM} storage for faster page loads. In *USENIX Conference on Web Application Development (WebApps 10)*, 2010.

[113] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.

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

[115] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[116] Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the web? the evolution of the web from a search engine perspective. In *WWW*, 2004.

[117] Anish Nyayachavadi, Jingyuan Zhu, and Harsha V Madhyastha. Characterizing "permanently dead" links on Wikipedia. In *IMC*, 2022.

[118] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[119] Seung-Taek Park, David M Pennock, C Lee Giles, and Robert Krovetz. Analysis of lexical signatures for improving information persistence on the world wide web. *ACM Transactions on Information Systems (TOIS)*, 22(4):540–572, 2004.

[120] Thomas A Phelps and Robert Wilensky. *Robust hyperlinks cost just five words each.* University of California, Berkeley, Computer Science Division, 2000.

[121] Mark E Phillips, Kristy K Phillips, and Sawood Alam. End of term web archive dataset: Longitudinal web archive of. gov and. mil domains. In *2023 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 98–101. IEEE, 2023.

[122] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

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

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

[125] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 677–694, 2011.

[126] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[127] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[128] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE working conference on mining software repositories (MSR 2010)*, pages 118–121. IEEE, 2010.

[129] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[130] Diomidis Spinellis. The decay and failures of web references. *Communications of the ACM*, 46(1):71–77, 2003.

[131] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. SpotSigs: Robust and efficient near duplicate detection in large web collections. In *SIGIR*, 2008.

[132] Dennis Trautwein, Aravindh Raman, Gareth Tyson, Ignacio Castro, Will Scott, Moritz Schubotz, Bela Gipp, and Yiannis Psaras. Design and evaluation of ipfs: a storage layer for the decentralized web. In *SIGCOMM*, 2022.

[133] Thomas Vissers, Wouter Joosen, and Nick Nikiforakis. Parking sensors: Analyzing and detecting parked domains. In *NDSS*, 2015.

[134] Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, Jason Wei, Chris Tar, Yun-Hsuan Sung, Denny Zhou, Quoc Le, et al. Freshllms: Refreshing large language models with search engine augmentation. *arXiv preprint arXiv:2310.03214*, 2023.

[135] Michele C Weigle, Michael L Nelson, Sawood Alam, and Mark Graham. Right html, wrong json: challenges in replaying archived webpages built with client-side rendering. In *2023 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 82–92. IEEE, 2023.

[136] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[137] Xinlei Yang, Wei Liu, Hao Lin, Zhenhua Li, Feng Qian, Xianlong Wang, Yunhao Liu, and Tianyin Xu. Visual-aware testing and debugging for web performance optimization. In *Proceedings of the ACM Web Conference 2023*, pages 2948–2959, 2023.

[138] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. Injecting mechanical faults to localize developer faults for evolving software. *ACM SIGPLAN Notices*, 48(10):765–784, 2013.

[139] Yifan Zhang, Chiu Tan, and Li Qun. Cachekeeper: a system-wide web caching service for smartphones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 265–274, 2013.

[140] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*, pages 14–24. IEEE, 2012.

[141] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

[142] Jingyuan Zhu, Anish Nyayachavadi, Jiangchen Zhu, Vaspol Ruamviboonsuk, and Harsha V Madhyastha. Reviving dead links on the web with fable. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, pages 131–144, 2023.

[143] Jingyuan Zhu, Huanchen Sun, and Harsha V Madhyastha. Toward better efficiency vs. fidelity tradeoffs in web archives. In *Proceedings of the 2025 ACM Internet Measurement Conference (IMC)*, 2025.

[144] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.

[145] Jonathan L Zittrain, John Bowers, and Clare Stanton. The paper of record meets an ephemeral web: An examination of linkrot and content drift within the new york times. *Available at SSRN 3833133*, 2021.

[146] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2019.