github and the crime against software

A software article by Efron Licht
May 2026

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

Introduction

> If a house was on fire there could be but two parties. One in favor of putting out the fire. Another in favor of the house burning.
>
> **Abraham Lincoln**

As I begin this article, Github is down, again. While there have been plenty of articles talking about github's struggles with reliability, security, and performance like this one by the verge, they are only scratching the surface. Github serves as a sort of 'signal' of infrastructural decay within both Github and the world of big tech software services in general. While I could point to similar decay in nearly every big tech corporate service, from AWS to Google Search, Github serves as a striking example due to being nearly syonymous with software development. I remember a conversation with a recruiter in 2022 where they simply did not believe that I was a 'real programmer' because I didn't have a github account.

What github is supposed to be - a high-performance, high-availability, high-capacity distributed system - is my personal professional specialty, so I can bring a little more insight into the problem than your average tech reporter.

It's not just availability: github is rife with problems. These include but are not limited to (parts in **bold** will be proven).

A small sample of recent problems with github

- **Github has dozens of incidents a month according to their own public communications**, which means the real rate is probably in the hundreds
- Github does not expose a public bug list or any issues page, hiding their problems deep in email chains
- **Github constantly lies to its users about it's uptime and priorities**
    - Even by its own metrics it is in violation of its SLA
    - **Github clearly prioritizes flashy AI features over fundamental reliability**
- **The increased 'agentic' load on github is the direct result of their own actions and those of their parent company, Microsoft, and that makes reliability problems their own fault**.
- Github often breaks on firefox and safari, browsers with millions of users
- Github tolerates clear 'follower-buying' and 'star-buying' on their services and promotes obviously astroturfed repositories: there is literally a website called githubstars.com that openly advertises buying stars, see this published paper by carnegie mellon, even though, quote, "fake stars are highly related to malicious activities"
    - Github's pull request page (and *especially* the comment/review page) are laggy and extraordinarily RAM-hungry (I've seen heap spikes of >512 MiB!)
- Github often resets or changes project and user settings: new features are 'on by default' even when extremely dangerous
- Github clearly prioritizes flashy 'AI' projects over actual paying customers
- Github Actions are badly designed, slow, and poorly documented, quite possibly the worst of the "shell scripts written in YAML" style of build systems and I've seen dozens
    - The github actions logs leak memory and have crashed my browser over multiple years, there's still no way to just pipe the stdout somewhere
    - The default cache behavior of github's 'default' actions (i.e, setup-go) are ludicrously naive and worse than nothing; many of them literally never do cache invalidation.
    - > I could go on and on. It is a fractal of bad design, to quote the famous rant about PHP.
- **Github's front-end…**
    - **is comically bloated and slow**
    - often breaks on safari and firefox
    - **constantly changes it's UX without warning or explanation, often deliberately replacing parts of the UX that used to do something useful with more and more funnels to their "AI" systems**

Github constantly lies to its users about it's uptime and priorities

While github's official status page claims something like 99.8% uptime - anyone who's used it recently knows the real number has zero nines in it. Sites like the missing github status page do a good job of showing what it really looks like.

"missing github status page", showing 87.06% availability

Github is well aware they're in hot water lately, and they've thrown out some press releases to try and deflect the criticism. Last month, github threw out a press release titled 'an update on github availability' to try and address some of this criticism.

> Microsoft: *The main driver is a rapid change in how software is being built. Since the second half of December 2025, agentic development workflows have accelerated sharply. By nearly every measure, the direction is already clear: repository creation, pull request activity, API usage, automation, and large-repository workloads are all growing quickly.*
>
> github's graph showing how many workloads they have

Beyond the fact that a graph with no labeled axes or individual data points is always a matter of suspicion, it is disengenous beyond belief to use the passive voice when saying "agentic development workflows have accelerated sharply", as though this is something that has snuck up on github without their knowledge or consent. Github is owned by microsoft, a company which has shoved AI, "agents" or otherwise, into every single product in a zillion overlapping ways. Almost every single github page has three

different AI buttons on it in the hotbar, two of which are dedicated entirely to starting agents. Many have four.

The increased 'agentic' load on github is the direct result of their own actions and those of their parent company, Microsoft

> Men are wisely presumed to intend the natural consequences of their conduct, and to seek what their acts seem to promote.
>
> *Charles Sumner, "The Crime Against Kansas"*

Github and microsoft are constantly pushing their users to use "AI" and "Agents" under every circumstance.

It is hard to overstate just how badly they want you to use this stuff: there are not one, not two, not three, but four different ways to fire up copilot just within the top right quarter of the landing page of your average repository. Four.

github landing page after login, with two different ai buttons

(While writing this article, I discovered there was a THIRD agent button on every page. For repos which project which don't explicitly disable agents, a FOURTH button shows up. This is diseased.)

github landing page after login, with ~~two~~ three ai buttons
repository page with FOUR DIFFERENT AI BUTTONS IN THE TOP RIGHT CORNER, THREE OF WHICH START AN AGENT ARE YOU KIDDING ME

Further, github subsized the cost of using these tools in order to drive adoption for years, effectively paying for a distributed denial of service on itself.

> Microsoft: *This exponential growth does not stress one system at a time. A pull request can touch Git storage, mergeability checks, branch protection, GitHub Actions, search, notifications, permissions, webhooks, APIs, background jobs, caches, and databases.* ***At high scale, small inefficiencies compound****: queues deepen, cache misses become database load, indexes fall behind, retries amplify traffic, and one slow dependency can affect several product experiences.*

Github's database engineers do not have an easy job - it is a legitimate challenge, one that I would struggle with myself - but Github does not have **small** inefficiencies, they have enormous, staggering inefficiencies, something we'll get into later.

> Microsoft: ***Our priorities are clear: availability first, then capacity, then new features.***

This is a lie. **Github - and the microsoft organization more widely - clearly prioritize flashy AI features over fundamental reliability** Github has a public changelog. In thirty days since they posted their update, their patch notes contain the

words “copilot” 59 times, “agent” 8 times, “performance” 0 times, and “reliability” 0 times. The changelog has a feature to filter by category: copilot is it’s own category: performance and reliability do not exist at all.

This is not just limited to github itself, it’s endemic cross the whole organization. Visual Studio Code, also owned by microsoft, is the most popular IDE (code editor) in the world, and is directly integrated with github and it’s “agentic” features, with more and more being shoveled in even as basic features like the built-in terminal degrade to near-unusability. As I opened VS Code this morning to work on my second draft, visual studio code pushed an update with a brand new “agent window”.

visual studio code update: ‘agents window’

Microsoft also included a section of the patch notes with “notable fixes”: it’s empty.

> notable fixes: none.

I am not blind. I am a human being. Don’t piss on my shoes and tell me it’s raining.

> Microsoft: *We are **reducing unnecessary work**, improving caching, isolating critical services, removing single points of failure, and **moving performance-sensitive paths into systems designed for these workloads**. This is distributed systems work: reducing hidden coupling, limiting blast radius, and making GitHub degrade gracefully when one subsystem is under pressure. We’re making progress quickly, but these incidents are examples of where there’s still work to do.*

This is an admission that the system is designed wrong. Performance is downstream of software architecture. If the bones are good, you can fix performance problems as they pop up: if the bones are bad, no amount of hacks will suffice: you’ll have to start over. The roots of Github’s reliability problems are on their back-end and within their databases, and these are hidden from us. However, we *can* see their front-end code - the HTML, Javascript, and CSS that are downloaded on to my computer or phone every time I visit their website. Let’s investigate this code to get an idea of the technical competence of github. If I see a dozen rats in a pizzeria’s dining room, it is theoretically possible their kitchen is spotless, but I’d be a fool to believe the owner about it. In a similar way, the obvious rot of github’s front-end indicates (but does not prove) larger problems in the back.

In the next section, we’ll take an experimental look at github’s front-end code to demonstrate the rot, comparing it directly to a commercial competitor, `gitlab`, and a free alternative, `codeberg`. (There are many others: I encourage you to repeat this experiment for yourself!)

github's front-end code is a mess: comparisons of github, gitlab, and codeberg.

Before you judge any code, first you have to know what it's trying to do. Most of github's front-end pages, and those of their competitors are essentially lists of links with some minor UX elements (buttons, tabs) and a search bar. There are very few expensive elements like interactive media or images.

These all look and function basically identically across these services (when they work), illustrated in the following screenshots.

> This is no surprise, as both codeberg and gitlab deliberately copied github's UX to be a familiar experience.

images: repository landing page for github, gitlab, codeberg

github landing page
gitlab landing page
codeberg landing page

They should be able to run cheaply and easily on basically any computer or cell phone with even a remotely good internet connection. In fact, github *used* to be able to do so - I distinctly remember using it on my first smartphone, an iPhone 4, over its 3G connection.

Assuming these provide the same features - and their feature sets are identical, at least for the bits I use - the 'best' code will be the one that uses the fewest resources (network bandwidth, CPU and clock time, RAM) and has the fewest bugs. While we can't know how many bugs github's frontend has, historical research has shown that the number of bugs is generally proportional to the number of lines. To quote Microsoft themselves:

> Industry average experience is about 1-25 errors per 1000 lines of code for delivered software. The software has usually been developed using a hodgepodge of techniques (Boehm 1981, Gremillion 1984, Yourdon 1989a, Jones 1998, Jones 2000, Weber 2003). Cases that have one-tenth as many errors as this are rare; cases that have 10 times more tend not to be reported. (They probably aren't ever completed!)
>
> The Applications Division at Microsoft experiences about 10–20 defects per 1000 lines of code during in-house testing and 0.5 defects per 1000 lines of code in released product (Moore 1992). The technique used to achieve this level is a combination of the code-reading techniques described in Other Kinds of Collaborative Development Practices, and independent testing.
>
> Steve McConnel, Code Complete, 2nd Ed (2004):

While I strongly doubt that github's release process has anything like the rigorous testing of Microsoft in the early 90s, let's be generous here and assume 1 defect/1000 lines.

Let's design an experiment to figure out the costs of these front-ends and estimate the potential bugs.

Experiment Design

Like any good experiment, we will want to minimize confounding variables.

- we'll throttle our internet speed to a "fast 3g" connection, both to minimize the impact of variance in networking conditions, and to simulate the experience of a github customer in less-than-ideal situations (say, in a rural area).
- we'll use an identical, minimal repository to compare all three to minimize the impact of repository complexity.
- We'll use the same browser (firefox) and computer (Apple Macbook Pro M5 Max with 48GiB of RAM) for all of the experiments.
- We will investigate four pages for each service:
    - the "landing" page, which shows the layout of the code
    - the "pull requests" page (sometimes called "merge requests"), which shows a summary of proposed changes by your coworkers
    - the "issues" page, which is self-explanatory
    - the "settings" page, ditto

For fun, we'll add my own website to the list. Obviously, what I do isn't the same as github, so this doesn't mean much.

Experiment Procedure

- Create a minimal git repository called `ghsucks` with a single branch (`master`), file (`README.md`), and zero issues, tags, or other cruft.
- Upload that git repo to each remote (`github`, `gitlab`, `codeberg`).
- For each page (landing, pull, issues, settings), with a clean cache
    - Collect a HTTP Archive (HAR) of network requests for analysis
    - Collect a heap snapshot after everything has loaded to get an estimate of 'steady-state' RAM usage
    - Collect information on compression support with `testcompress`
- Analyze the data
    - Analyze the HAR `anhar`, a small program I wrote for this
    - Analyze compression support with `testcompress`, a small program I wrote for this.
- Use a third-party service, `pagespeed.web.dev` to analyze the sites as well to see if their conclusions line up with ours. Always good to have a scond opinion.

> There's plenty more you could do to compare this front-end code: a good extension to this experiment would be digging in to a profiler and finding which segments of the code are actually executed and how often, what's blocking the initial paint of the screen, what takes longest to paint, etc. I have spent far too long on this article already and will leave it as an exercise to the reader (or better yet, the developers who actually maintain these services).

running the experiment

Creating the repo

```
#!/usr/bin/env bash

# create the repo...
mkdir ghsucks
cd ghsucks
git init

# create a file
echo '# ghsucks

example repository for the purpose of Efron Lichts article "Github and the Crime Against Software"' > README.md
# add it to the repo
git add README.md

# create the commit
git commit -m 'initial (and only) commit'
```

Uploading the repo to `github`, `gitlab`, and `codeberg`

```
#!/usr/bin/env bash

git remote add github https://github.com/ef0xa/ghsucks && git push github master
git remote add gitlab https://gitlab.com/efronlicht/ghsucks && git push gitlab master
git remote add codeberg https://codeberg.org/efronlicht/ghsucks && git push codeberg master
```

Collecting network sample data

Sadly, this is not easily automated. Here's the (manual) algorithm I used, with examples.

load the page in firefox and open the inspect tool

right-click on the page and select the 'inspect' panel

throttle the internet connection to the "fast 3G" preset and disable cache to simulate a fast cellular network connection on an `iPhone 4` in 2012 (the cell phone I had when I first used github).

disable the cache and throttle to 'fast 3g'

reload the page, then save the `har` archive to disk

save the HAR archive

Collect heap (ram) usage after load:

Collecting heap snapshots is also neat easily automated, at least as far as I know. Same deal as before: use firefox's built in profiling tool, this time under the "memory" tab, and take a heap snapshot:

what	usage	image
github	69 MiB	github uses 69 MiB
gitlab	68 MiB	gitlab uses 68 MiB
codeberg	14 MiB	codeberg uses 14 MiB
eblog.fly.dev	1.3 MiB	eblog.fly.dev uses 1.3 MiB

Analysis of heap usage and historical comparisons

Both Github and Gitlab use an inexcusable amount of RAM even when they're not doing anything. For example, the playstation 2 32 MiB of RAM in total and was capable of rendering 3d graphics, decoding audio, and handling multiple sources of IO simultaneously. The nintendo wii had only 24MiB of 'regular' RAM and was able to run 3d games and render streaming video.

Worse yet, this is actually understating the problem - the usage quickly spikes on pages which have actual content. For example, a heap snapshot of `https://github.com/rust-lang/rust/pulls` used 148MiB of RAM *just to render the first page of issues*, more than the original iPhone had. This is a **disgusting** level of waste for a text-only page with a handful of links. I have no idea what they could possibly be doing to waste so much RAM, but they need to stop.

[Table: historically important or interesting computers and their amount of RAM]

manufacturer	device	release date	total memory	note
apple	iphone 1	2007	128 MiB	
apple	iphone 4	2010	512 MiB	
sony	playstation 1	1994	2.5 MiB (2MiB base, 512 KiB gpu)	capable of full-3d gameplay
sony	playstation 2	2000	32 MiB	powerful 3d graphics
ibm	PC	1981	16-256 KiB	yes, this is where the name 'PC' came from
nintendo	gameboy	1989	16 KiB (8 KiB CPU, 8KiB video)	
nintendo	wii	2006	88 MiB (24 MB 1T-SRAM + 64 MB GDDR3 SDRAM)	
DEC	PDP-11	1970	~56 KiB	Origin of UNIX, able to run multiple user session, networking, I/O, etc.
NASA	apollo 11 guidance computer	1969	~2KiB	Literally went to the moon

---

Use `testcompress` to collect information on compression support

Compression is a valuable way to lower the load for the client and server and everyone in-between. `gzip` is the tried and true compression standard and I expect every website to support it. `zstd` has favorable performance characteristics but is more modern: supporrt for it is a 'nice-to-have' but not a big deal.

In my comparisons, I'm using the maximum compression levels in go's standard library `gzip` and the `github.com/klauspost/compress/zstd` library for comparison to give an idea of what the peak savings could look like: the actual level of compression used by the servers may differ but generally doesn't differ too much.

IN

```
#!/usr/bin/env bash
go run ./testcompress \
    'https://eblog.fly.dev/startingsystems3.html' \
    'https://github.com/ef0xa/ghsucks' \
    'https://gitlab.com/efronlicht/ghsucks' \
    'https://codeberg.org/efronlicht/ghsucks'
```

URL	MIME	supported encodings	Bytes	Bytes (gzip)	Bytes (zstd)	Ratio (gzip)	Ratio (zstd)
eblog.fly.dev/startingsystems3.html	text/html; charset=utf-8	zstd gzip	575.77 KiB	67.64 KiB	60.17 KiB	11.75%	10.45%
github.com/ef0xa/ghsucks	text/html; charset=utf-8	gzip	224.70 KiB	43.27 KiB	37.96 KiB	19.26%	16.90%
gitlab.com/efronlicht/ghsucks	text/html; charset=utf-8	gzip	43.08 KiB	11.48 KiB	11.34 KiB	26.64%	26.32%
codeberg.org/efronlicht/ghsucks	text/html; charset=utf-8	n/a	43.88 KiB	13.00 KiB	12.79 KiB	29.62%	29.16%

Analyzing the network archive (HAR) with `anhar`

I wrote a small program , `anhar` (**an**alyze **har**) to analyze this dump. Anhar parses the JSON, processes the responses by auto-formatting HTML, CSS, and JavaScript with `deno fmt`, and prints out a summary of the results. The source code and full results are attached in the 'bonus content' section of this article.

> We auto-format the code because github uses a minifier on its front-end code before transmission. There's nothing wrong with this, but it makes it hard to get an estimate of the number of lines. I picked 'deno' out of a hat - try it using your own formatter if you think it gives a bad estimate.

IN

```
#!/usr/bin/env bash
for dir in dumps/*; do
  go run ./cmd/anhar -output-format markdown $dir/* > "$(basename $dir).dump.md"
done
wait
cat *.dump.md > all.dump.md
```

The output looks like this, split up by kind of HTTP request:

MIME	URL	Count	Request Size	Response Size (minified)	Response Size (expanded)	Response Lines	Response Lines (expanded)	Cont Load
`image/svg+xml`	https://github.com/ef0xa/ghsucks/settings	2	1020 B	3 KiB	3 KiB	8	30	778m
`text/plain`	https://github.com/ef0xa/	10	27 KiB	6 KiB	6 KiB	10	10	778m

MIME	URL	Count	Request Size	Response Size (minified)	Response Size (expanded)	Response Lines	Response Lines (expanded)	Cont Load
	ghsucks/settings							
image/png	https://github.com/ef0xa/ghsucks/settings	2	2 KiB	53 KiB	53 KiB	2	2	778m
application/json; charset=utf-8	https://github.com/ef0xa/ghsucks/settings	7	12 KiB	53 KiB	54 KiB	9	96	778m
text/html; charset=utf-8	https://github.com/ef0xa/ghsucks/settings	1	1 KiB	293 KiB	293 KiB	3425	3425	778m
text/css	https://github.com/ef0xa/ghsucks/settings	39	17 KiB	1 MiB	2 MiB	79	55035	778m
application/javascript	https://github.com/ef0xa/ghsucks/settings	194	81 KiB	11 MiB	17 MiB	1489	427469	778m
/	https://github.com/ef0xa/ghsucks/settings	255	144 KiB	14 MiB	20 MiB	5022	486067	778m

Explanation of Sections

Name	Explanation
MIME	kind of thing (i.e, file, data stream, etc) sent in the response.
URL	where we got it from
Count	how many request/response pairs per MIME/URL we have
Request Size	total size of the HTTP requests sent, estimated by adding up the HTTP Headers.

Name	Explanation
Response Size (minified)	Number of bytes actually received by my computer, including the headers and body.
Response Lines (minified)	Number of newline characters in the responses as actually received. Not very useful.
Response Size (expanded)	Number of bytes after the response has been put through `deno fmt`.
Response Lines (expanded)	Number of newline characters in the response after it's been put through `deno fmt`. A decent estimate for the number of lines in the original code.
Content-Load	Time to load the HTML page that serves as the "base" of the page. For regular HTML websites, like the one you're reading right now, this is how long it takes for the site to be readable. Unfortunately, github uses react for it's frontend, so it's unusable until...
Load	Time to load all the content requested by the 'base' HTML page. Even when this is done, the page may not be ready to use: javascript may need to execute, but it's a good estimate.

The breakdown by MIME is handy, but it's also a bit noisy, so let's just look at the totals for now:

```
#!/usr/bin/env bash
cat all.dump.md | rg --fixed-strings '*/*'
```

MIME	URL	Count	Request Size	Response Size (minified)	Response Size (expanded)	Response Lines	Response Lines (expanded)	Conte Load
/	https://codeberg.org/efronlicht/ghsucks	11	9 KiB	1 MiB	1 MiB	1403	3480	2.934s
/	https://codeberg.org/efronlicht/ghsucks/issues	9	7 KiB	1 MiB	1 MiB	1409	354	3.394s
/	https://codeberg.org/efronlicht/ghsucks/pulls	11	9 KiB	1 MiB	1 MiB	1578	1869	2.682s
/	https://codeberg.org/efronlicht/ghsucks/settings	12	9 KiB	1 MiB	1 MiB	1616	4054	2.857s
—	—	—	—	—	—	—	—	—
/	https://eblog.fly.dev/index.html	5	2 KiB	196 KiB	196 KiB	404	403	67ms
/	https://eblog.fly.dev/startingsystems3.html	5	2 KiB	766 KiB	766 KiB	3875	3874	76ms

MIME	URL	Count	Request Size	Response Size (minified)	Response Size (expanded)	Response Lines	Response Lines (expanded)	Conte Load
—	—	—	—	—	—	—	—	—
/	https://github.com/ef0xa/ghsucks	291	178 KiB	15 MiB	22 MiB	4229	544564	843m
/	https://github.com/ef0xa/ghsucks/pulls	266	151 KiB	14 MiB	20 MiB	2883	487922	592m
—	—	—	—	—	—	—	—	—
/	https://github.com/ef0xa/ghsucks/settings	255	144 KiB	14 MiB	20 MiB	5022	486067	778m
/	https://gitlab.com/efronlicht/ghsucks/-/boards	57	151 KiB	7 MiB	5 MiB	9781	58663	6.621
/	https://gitlab.com/efronlicht/ghsucks	78	196 KiB	8 MiB	7 MiB	10031	101682	6.88s
/	https://gitlab.com/efronlicht/ghsucks/-/merge_requests	65	159 KiB	7 MiB	6 MiB	9885	90567	6.947
/	https://gitlab.com/efronlicht/ghsucks/edit	117	285 KiB	7 MiB	6 MiB	10224	71916	6.964

It is immediately obvious that github is doing orders of magnitude more work than its competitors. To wit:

1. First and foremost, there is *too much code*, entirely out of proportion to the problem it's supposed to solve. There is more code pulled to display an empty repository (540K) than it took to build DOOM (35K of mostly C over 150 files), Windows Quake (120K lines over 261 files, mostly C), the entire MS-DOS 4.0 operating system (332K lines, including a bunch of platform-specific assembler), or the entire source code of the zig toolchain (460K over 170 files, all zig, not including the standard libraries).

2. Even if there *were* some excuse for the amount of code, the code is split among hundreds of files, multiplying the overhead. There is no reason for any individual page to pull in more than one CSS file: this has 40. There is no reason for any website to have more than one javascript file: Github splits their code across hundreds for no conceiveable reasons.

    > Actually, I know why - they've used webpack to split the javascript into chunks. In theory, this lets you split out 'unnecessary' javascript from 'core' functionality and only load chunks loaded when actually required. If this is the 'necessary' javascript, the 'optional' javascript

must be the size of the sun. IIn theory this could be nice for caching and CDN if the different sub-sections of their site have overlapping but not identical dependencies: in practice, all this does is slow down the load - every single file requires an independent HTTP request, adding extra overhead. (If you're a programmer interested in learning more about the structure of HTTP, my series of articles, Backend From The Beginning, might be a fun way to look.)

3. It is **too slow!** Waiting **22 seconds** just to view a blank page is a sick joke. Even for fast computers and faster connections the behavior is unacceptable. My home internet is a fiber connection with over a GiB/s down and my computer has the fastest consumer processor in the world. It still takes over a second for the repository to be remotely usable. I shudder to think how this performs for people who can't afford a $4000 computer and a fiber connection (i.e, the vast majority of software developers).

This is deeply embarrassing, not the result of a mere miscalculation or a few months' worth of technical debt but a historic boondoggle up there with the Mark 14 Torpedo. This is the kind of horrific mistake you can only get from years of bad programming, a rampant addiction to LLM code vomit, and a deeply disconnected if not intentionally malicious technical leadership, or more likely, all of the above. No one would accept a seven-ton frying pan or an obese racing horse, at least not for a practical purpose - you'd be laughed out of the room. Yet we accept flagrantly unsuitable software with nary a thought. What I am asking here - what should be. the bare minimum - is github's performance as it *actually existed* in 2012.

picture of a very fat horse with the label "small inefficiencies in github frontend"
picture of "the world's largest frying pan"

Lemma: It's not mere enshittification

The common term for what's happening to Github and other software infrastructure products is "enshittification". You're probably familiar with the term, but to briefly summarize:

1. a good product or service is made that is good for users and business customers
2. they abuse the users to make things better for the business customers
3. they abuse the business customers to make things better for themselves

Microsoft and Github are no strangers to this process - or to it's predecessor, Embrace, Extend, Extinguish but that's not quite what's happening here. The problems here are not merely bad for Github's users and business customers, they are bad for Microsoft. There is no conceivable gain to Github to turn their codebase into a hoarder's nest: *they are the ones who pay the costs of this extra load*, both directly - in terms of server and bandwidth costs - and indirectly in the term of untold millions of engineer-hours it takes to keep their ramshackle, bloated codebase limping along. This is something

beyond enshittification - it's like a kind of slow suicide to appease the delusions of executives and venture capitalists, or maybe a kind of performance art.

> And yes, it is millions. Github appears to have about 800 engineers. Assuming they work 40-hour weeks 48 weeks a year, that's 1,536,000 engineer-hours a year.

I'm not so good at this business psychology stuff - go to my friend Nikhil Suresh for that - so let's get back to the facts. In the next section, we'll do a more detailed comparison of github, gitlab, codeberg, and `eblog.fly.dev`'s front-end, combined with some practical suggestions for their developers (on how to improve their front-end and it's delivery) and users who have to cope with them.

Third-party analysis: pagespeed.web.dev

Maybe I'm being a little too harsh. Let's see what `pagespeed.web.dev` has to think about them:

Mobile

what	first contentful paint	largest contentful paint	interaction to next paint	assessment	image
codeberg	1.2s	1.3s	86 ms	PASS	codeberg-pagespeed
eblog.fly.dev	1.1s	1s	N/A	PASS	eblog pagespeed
github	1.8s	2.1s	242 ms	FAIL	github-pagespeed
gitlab	2.1s	2.4s	243 ms	FAIL	gitlab-pagespeed

Pagespeed has great recommendations on how to improve performance: I'll highlight some of the more interesting ones and my own thoughts as we go along.

Comparison of `github`, `gitlab`, and `codeberg`.

(Script evaluation and Parsing & Compilation metrics provided by pagespeed.web.dev)

gitlab

Summary

- content-load ~7s, load ~13s
- 7MiB over 70 files and ~10_000 lines
- Estimated bugs: 10.
- steady-state heap: ~68MiB
- Supports `gzip` but not `zstd`.
- Script Evaluation: 1593ms
- Script Parsing & Compilation: 492 ms

Grade: `D+`

Gitlab is struggling in class and will not be able to pass if it continues at this rate, but not all hope is lost.

While gitlab isn't quite as spendthrift as github, it still pulls far too many resources and uses them badly, pulling in over a megabyte of javascript and CSS that are never even touched. The code that *is* used is awful, including a 3-megabyte chunk which takes 255 milliseconds to even *parse*, much less execute. It also parses and executes all of that serially, making a bad problem worse. This is enough to convince me to move off gitlab, which seems just as incompetent. Also striking is over 200ms spent on garbage collection before the page even loads: this is amateur hour stuff.

This is at least a small enough number of files that it's possible to read that code and fix. The first place to start is by eliminating `supersidebar.js` with extreme prejudice.

Suggestions (developer)

- First and foremost, get rid of most of this code. There's no need for 55000 lines of CSS for a landing page: this suggests a fundamental incoherence. My instinct tells me you could get away with the tenth of the line of both CSS and Javascript: dig into the code and cut it down to the bone.
- There's no need for any of the plain-text files: combine them into the HTML.
- Once you've cut down the weight, minimize the number of requests to get better speed. The best way to do this is just to have a single file each for CSS, HTML, and Javascrfipt: the "text/plain" ones should be inlined into the HTML.
- Further minifying the JS and CSS couldn't hurt, but that's a "nice to have".

Suggestions (user)

Gitlab also seems to be degrading, if not at the same rate as github. If you are a gitlab user, my advice is not to tie yourself to anything gitlab-specific in their toolchain.

Codeberg/forgejo

Summary

- Content-Load 2.9s, Load 3.1s - the HTTP is uncompressed and sent rather slowly.
- Estimated bugs: 2.
- Steady-state heap: ~14MiB.
- 1MiB over 11 files and ~1100 lines.
- Neither `gzip` or `zstd` are supported.

Grade: C+

`codeberg/forgejo` is a promising young student with a good grasp of the fundamentals, but their lack of attention to detail and inexperience show. For example, they don't minify their HTML (a small detail) **and don't support compression**, a big miss. Additionally, the vast majority of their javascript and CSS are is not necessary to render the page (good) but are blocking the page render (bad). If they can buckle down and focus on the fundamentals I think they can get a B or A by the end of the year.

Suggestions (developer)

Even though this is much smaller than it's competitors, it's still comically bloated.

- combine the javascript and CSS payloads to have fewer requests.
- support gzip and zstd compression for HTTP payloads, especially HTML, to have faster initial loads: feel free to use my efronlicht/compresshttp library. For static content, a nice trick is to store the content on disk pre-compressed so you don't have to do any work on your side: that's how this website works.
- minify HTML as well as the CSS and JSS. to shave some bytes (less necessary if you compress, but everything helps).
- **Take the knife to this CSS file and this javascript file, which are both comically large and mostly unused**, and then defer the evaluation of both, and you'll have a real smooth experience.

Suggestions (user)

- codeberg/forgejo seem promising, but the hosting may not be reliable since it relies on independent donations, and the quality of the programming seems

somewhat naive. Unsurprising - it's a product provided by free volunteers - but still a better bet than the awful corporate monstrosities it's competing with. Happily, since it's free software, you can always migrate from that to another instance: for example, self-hosting. try it out but have a corporate backup if you need it.

Commentary:

There's still plenty of room to improve - a 3s load is far too slow - but browser caching will help you there, and compression and bundling of the remaining code could get them the rest of the way. Compared to the rot of the rest of the industry, this is a breath of fresh air.

github

Summary

- ~300 files @ 550_000 lines of code and data (yes, really!)
- Estimated bugs: 550.
- ~800ms content-load
- ~7s full load
- steady-state heap of ~69mib
- Supports `gzip` but not `zstd`.
- Script Parsing & Compilation: 241 ms
- Script Evaluation: 429 ms
- Style & Layout: 499ms

Grade: F.

It is clear this student is using LLMs to do their homework.

Commentary

Hoo boy, where do I even start. Beyond the enormous weight, github always fetches every theme on every page regardless of whether you use them!

Suggestions (developer)

This is code is comically overweight that your best approach might be muntzing: start ripping out parts at random until your tests fail. If they don't, assume the code is

worthless. If an executive tries to put a fifth AI button in there, ~~shoot them~~ politely decline.

`pagespeed.web.dev` suggests that 528KiB of javascript and CSS is **literally completely unused**, so you could start there.

Suggestions (user)

Run while you still can. If you're forced to stay on github for whatever reason, they are in violation of their own SLA, and I strongly suggest you file a support ticket to get your refund.

eblog.fly.dev/startingsystems3.html

Summary

- content-load 76ms, load 1.1s
- 766 KiB over 5 files and ~3800 lines. 576KiB of these are the single HTML file, which compresses down to ~70KiB via zstd.
- Steady-state heap: ~11MiB
- Supports both `zstd` and `gzip`.

Grade: A-

`eblog.fly.dev` is pretty good, but the HTML seems bloated and repetitive, even with the compression, and there are five requests needed to load the page when it could just be one. The high heap cost (well, high from my point of view) is probably due to the repetitive inline literals; it's worth investigating.

> pagespeed.web.dev gives me a perfect score, but I'm not so convinced. I'm good but hardly perfect.
>
> efron's perfect score

Suggestions (developer)

- Move some of the repetitive formatting into a handful of styles instead of doing them inline.
- Consider dropping the custom fonts or using smaller ones to cut down on load; definitely set them to 'swap' or 'optional' not to block page loads to make fonts look slightly nicer. (Done as of this article's publication.)
- Try inlining the CSS inside the document to remove the synchronous dependency would speed up the gap between full load.

> Yes, this is advice from me to me.

Suggestions (user)

- Try telling Efron he did a great job. Boy, it sure seems like he knows a lot about high-performance, reliable software: if you're in the tech business, he has a fantastic record of developing software on-time and under-budget, and has saved clients and employers millions of dollars: He is available for short and long term contracts, advice, hiring support, etc etc etc: reach out today for a free consult. You might as well sign up for a free consult: I don't charge we're both convinced I can solve your problem.
- Don't just take my word for it:

> *I have extensive exposure to programmers that are better than me and people that are smarter than me. Every Thursday, I have a call with Efron Licht, and frankly I can scarcely grasp why someone that competent spends time talking to me... I'm not competing with Efron. If I was, I'd either have to study for five hours every day for the rest of my life, or shut the company down tomorrow.*
>
> **Nikhil Suresh** (who gives himself too little credit, but I'll take the plug.)

- You can also check out my other articles here on eblog.fly.dev. My personal favorite is one about the historical wisdom of engineers and programmers of the 50s-80s, called "software talmud".

Ok, enough self-promotion.

Conclusion:

> Crime is always suspicious, and shakes, like a sick man, merely at the pointing of a finger.
>
> **Charles Sumner, "The Crime Against Kansas"**

It is increasingly clear that the large corporations simply cannot be trusted with software infrastructure - like Darth Vader, they will alter the deal at their own convenience and shrug off the petty complaints of us mere mortals. It's not just github - the whole internet is a diseased, and github is just a particularly disgusting outbreak.

It seems as though the entire industry has forgotten that software is supposed to solve problems for users. They whine and cry and bluster whenever you bring it up, as though software bugs and bad performance are merely a petty complaint. This is just not true.

We live in a world submerged in software and we're drowning with it - we need software to do almost anything in modern society. We have the curious dilemma that the services built and maintained by the richest and most powerful companies in the world, with legions of highly-paid engineers, are obviously rickety garbage, easily outperformed by the personal projects of small teams of unpaid amateurs.

Most of these frontend problems could be fixed - or at least mitigated - in under a week by a halfway competent engineer who literally just followed the recommendations on pagespeed. Hell, they could probably point their much-vaunted AI tools at it. They either do not care, are ludicrously incompetent, or are so hamstrung by AI-mad, investor-mad leadership that they are actively prevented from fixing low-hanging fruit that would *save them money*.

It seems to me that we have elevated a group of people to prominence who lack even a basic level of competence or professional integrity. We would not accept Microsoft suddenly dumping thousands of tons of chemical runoff into our drinking water, but we accept it in our internet infrastructure.

> Actually, this comparison is not even abstract: the overwhelming increase in datacenter demand due to our AI bubble is causing a huge increase in discharged groundwater. In theory, it's supposed to be properly treated before being released back into the environment, but if their environmental engineering is anything like their software engineering..

I do not mean to be a negative person. I love software and programming. Software's my life and my wife, to misquote the velvet underground. The power of software - to think about a problem and solve it perfectly forever - is a wonderful gift that most generations could have scarcely dreamed of. The beauty of software is that if you write something correctly once, you can replicate it perfectly, forever, for free, that everyone can use. The wretched waste and incompetence of Github and similar services is not merely a bad product, it is a **crime against software.**

Appendix

anhar source

```
 1  package main
 2
 3  import (
 4      "bytes"
 5      "context"
 6      "encoding/json"
 7      "flag"
 8      "fmt"
 9      "io"
10      "log"
11      "maps"
```

```
12      "os"
13      "os/exec"
14      "path"
15      "path/filepath"
16      "slices"
17      "strings"
18      "sync"
19      "text/tabwriter"
20      "time"
21
22      "github.com/klauspost/compress/zstd"
23  )
24
25  type (
26      // Request is a simplified version of HAR Entry.
27      Request struct {
28          Method,
    URL
    string
29          RequestBodySize, ResponseBodySize,
    ExpandedResponseHeaderSize int
30          RequestHeadersSize,
    ResponseHeadersSize                    int
31          ResponseLines,
    ExpandedResponseLines                        int
32
    TotalRequestBytes
    int
33
    TotalResponseBytes
    int
34
    TotalBytes
    int
35      }
36
37      Sizes struct { // aggregate sizes for a given MIME
    type
38          Count                                int
39          RequestSize, ResponseSize, TotalSize int
40          ResponseLines, ExpandedResponseLines int
41          ExpandedResponseSize                 int
42      }
43
44      Summary struct { // Result of processing an ANHAR.
45          Title              string
46          ContentLoad, Load time.Duration
47          TotalRequests      int
48          TotalSize          int
49          Requests           []Request
50          ByMime             map[string]Sizes
51      }
52      Header struct{ Name, Value string }
53      ANHAR  struct { // Subset of HAR.
```

```
54          Log struct {
55              Version          string
56              Creator, Browser struct{ Name, Version
    string }
57              Pages            []struct {
58                  PageTimings struct {
59                      OnContentLoad, OnLoad int
60                  }
61                  Title string
62              }
63              Entries []struct {
64                  Request struct {
65                      Method, URL string
66                      BodySize    int
67                      HeadersSize int
68                      Headers     []Header
69                  }
70                  Response struct {
71                      Status  int
72                      Headers []Header
73                      Content struct {
74                          MimeType string
75                          Size     int
76                          Text     string
77                      }
78                  }
79              }
80          }
81      }
82  )
83
84  var tmp, _ = os.MkdirTemp("", "dumphar-*")
85
86  var dir = flag.String("dir", tmp, "directory to write
    formatted files to")
87  var outputformat = flag.String("output-format", "tab",
    "output format: tab or markdown")
88
89  func main() {
90      log.SetFlags(log.Lshortfile)
91      flag.Parse()
92      switch *outputformat {
93      case "markdown", "tab":
94      default:
95          log.Fatalf("unknown output format: %s",
    *outputformat)
96      }
97
98      var archives []io.Reader
99      for _, f := range flag.Args() {
100         f, err := os.Open(f)
101         if err != nil {
102             log.Fatal(err)
103         }
```

```
104             if filepath.Ext(f.Name()) == ".zst" {
105                 r, err := zstd.NewReader(f)
106                 if err != nil {
107                     log.Fatal(err)
108                 }
109                 archives = append(archives, r)
110             } else {
111                 archives = append(archives, f)
112             }
113         }
114
115         summaries := make([]Summary, len(archives))
116         errs := make([]error, len(archives))
117         wg := new(sync.WaitGroup)
118         for i := range archives {
119             wg.Go(func() {
120                 subdir := fmt.Sprintf("%s/%d", *dir, i)
121                 os.MkdirAll(subdir, 0744)
122                 var anhar ANHAR
123                 err :=
    json.NewDecoder(archives[i]).Decode(&anhar)
124                 if err != nil {
125                     log.Printf("read %s: %v", flag.Arg(i),
    err)
126                     return
127                 }
128                 summaries[i], errs[i] = Summarize(&anhar,
    *dir)
129                 if err != nil {
130                     log.Printf("summarize %s: %v",
    flag.Arg(i), err)
131                     return
132                 }
133             })
134         }
135         wg.Wait()
136         // print load times and totals
137         switch *outputformat {
138         case "markdown":
139             for i, s := range summaries {
140                 if errs[i] == nil {
141                     writeMarkdownSummary(os.Stdout, s)
142                 }
143             }
144         case "tab":
145             for i, s := range summaries {
146                 if errs[i] == nil {
147                     writeTabSummary(os.Stdout, s)
148                 }
149             }
150         default:
151             log.Fatalf("unknown output format: %s",
    *outputformat)
152         }
```

```
153
154  }
155  func writeMarkdownSummary(w io.Writer, s Summary) {
156      fmt.Fprintf(w, "## %s\n\n", s.Title)
157      fmt.Fprintf(w, "- content-load: %s\n", s.ContentLoad)
158      fmt.Fprintf(w, "- load: %s\n\n", s.Load)
159
160      fmt.Fprintf(w, "| MIME | Count | Request Size |
     Response Size (minified) |  Response Size (expanded) |
     Response Lines | Response Lines (expanded) |\n")
161      fmt.Fprintf(w, "| --- | --- | --- | --- | --- | --- |
     --- |\n")
162
163      keys := slices.Collect(maps.Keys(s.ByMime))
164      slices.SortFunc(keys, func(a, b string) int {
165          switch {
166          case a == "*/*" && b != "*/*":
167              return 1
168          case a != "*/*" && b == "*/*":
169              return -1
170          default:
171              return s.ByMime[a].TotalSize -
     s.ByMime[b].TotalSize
172          }
173      })
174      for _, k := range keys {
175          v := s.ByMime[k]
176          fmt.Fprintf(w, "| `%s` | %d | %s | %s | %v | %v |
     %d |\n", k, v.Count, fmtBytes(v.RequestSize),
     fmtBytes(v.ResponseSize),
     fmtBytes(v.ExpandedResponseSize), v.ResponseLines,
     v.ExpandedResponseLines)
177      }
178      fmt.Fprintln(w, "")
179  }
180
181  func writeTabSummary(w io.Writer, s Summary) {
182      const (
183          minwidth = 4
184          tabwidth = 4
185          padding  = 3
186          padchar  = ' '
187          flags    = 0
188      )
189      fmt.Fprintln(w, "")
190      fmt.Fprintf(w, "GET %s: content-load %s, load
     %s\n\n", s.Title, s.ContentLoad, s.Load)
191
192      tw := tabwriter.NewWriter(w, minwidth, tabwidth,
     padding, padchar, flags)
193      format := strings.Repeat("%v\t", 7) + "%s\n"
```

```
194         fmt.Fprintf(tw, format, "FILE", "MIME", "Count",
    "Request Size", "Response Size (minified)",
    "Response Size (expanded)", "Response Lines", "Response
    Lines (expanded)")
195
196         defer tw.Flush()
197
198         keys := slices.Collect(maps.Keys(s.ByMime))
199         slices.SortFunc(keys, func(a, b string) int {
200             switch {
201             case a == "*/*" && b != "*/*":
202                 return 1
203             case a != "*/*" && b == "*/*":
204                 return -1
205             default:
206                 return s.ByMime[a].TotalSize -
    s.ByMime[b].TotalSize
207             }
208         })
209         for _, k := range keys {
210             v := s.ByMime[k]
211             fmt.Fprintf(tw, format, s.Title, k, v.Count,
    fmtBytes(v.RequestSize), fmtBytes(v.ResponseSize),
    fmtBytes(v.ExpandedResponseSize), fmt.Sprintf("%d",
    v.ResponseLines), fmt.Sprintf("%d",
    v.ExpandedResponseLines))
212         }
213         fmt.Fprintln(w, "")
214
215     }
216
217     func calcHeaderSize(headers []Header) int {
218         // header: Key: Value\r\n
219         var sum int
220         for _, h := range headers {
221             sum += len(h.Name) + len(h.Value) + len(": \r\n")
222         }
223         return sum
224     }
225
226     var units = []string{"B", "KiB", "MiB", "GiB", "TiB",
    "PiB", "EiB", "ZiB", "YiB"}
227
228     func fmtBytes(n int) string {
229         for _, u := range units {
230             if n < 1024 {
231                 return fmt.Sprintf("%v %s", float64(n), u)
232             }
233             n /= 1024
234         }
235         panic("too big")
236     }
237
```

```
238  // Summarize an archive. This will write the response
     bodies to disk and call a formatter to get a better sense
     of how much code it "really" is.
239  func Summarize(a *ANHAR, dir string) (Summary, error) {
240      requests := make([]Request, len(a.Log.Entries))
241      if err := os.MkdirAll(dir, 0744); err != nil {
242          return Summary{}, err
243      }
244      byMime := make(map[string]Sizes)
245      // write the files to disk and call a formatter to get
     a better sense of how much code it "really" is.
246      args := make([]string, 1, len(a.Log.Entries)+1)
247      args[0] = "fmt"
248      for _, e := range a.Log.Entries {
249          f, err := os.Create(fmt.Sprintf("%s/%s", dir,
     path.Base(e.Request.URL)))
250          if err != nil {
251              return Summary{}, err
252          }
253          f.WriteString(e.Response.Content.Text)
254          f.Sync()
255          f.Close()
256          args = append(args, f.Name())
257      }
258      if out, err := exec.CommandContext(context.TODO(),
     "deno", args...).CombinedOutput(); err != nil {
259          log.Printf("%s: %s", err, out)
260      }
261
262      var sizes []int
263      var lines []int
264      for i := range args[1:] {
265          b, err := os.ReadFile(fmt.Sprintf("%s/%s", dir,
     path.Base(a.Log.Entries[i].Request.URL)))
266          if err != nil {
267              return Summary{}, err
268          }
269          sizes = append(sizes, len(b))
270          lines = append(lines, bytes.Count(b, []byte{'\n'})
     +1)
271      }
272      for i, e := range a.Log.Entries {
273
274          reqHeadersSize :=
     calcHeaderSize(e.Request.Headers)
275          respHeadersSize :=
     calcHeaderSize(e.Response.Headers)
276          requests[i] = Request{
277              Method:                    e.Request.Method,
278              URL:                       e.Request.URL,
279              ResponseBodySize:
     max(e.Response.Content.Size,
     len(e.Response.Content.Text)),
```

```
280                 RequestBodySize:
    e.Request.BodySize,
281                 RequestHeadersSize:          reqHeadersSize,
282                 ResponseHeadersSize:         respHeadersSize,
283                 ResponseLines:
    strings.Count(e.Response.Content.Text, "\n") + 1,
284                 ExpandedResponseLines:       lines[i],
285                 ExpandedResponseHeaderSize: respHeadersSize +
    sizes[i],
286                 TotalRequestBytes:
    e.Request.BodySize + reqHeadersSize,
287                 TotalResponseBytes:
    max(e.Response.Content.Size,
    len(e.Response.Content.Text)) + respHeadersSize,
288                 TotalBytes:
    max(e.Response.Content.Size,
    len(e.Response.Content.Text)) + respHeadersSize +
    e.Request.BodySize + reqHeadersSize,
289             }
290             requests[i].TotalBytes =
    requests[i].TotalRequestBytes +
    requests[i].TotalResponseBytes
291             { // update MIME summary
292
293                 v := byMime[e.Response.Content.MimeType]
294                 v.Count++
295                 v.RequestSize += requests[i].TotalRequestBytes
296                 v.ResponseSize +=
    requests[i].TotalResponseBytes
297                 v.TotalSize += requests[i].TotalBytes
298                 v.ExpandedResponseLines +=
    requests[i].ExpandedResponseLines
299                 v.ResponseLines += requests[i].ResponseLines
300                 v.ExpandedResponseSize += int(sizes[i]) +
    respHeadersSize
301                 byMime[e.Response.Content.MimeType] = v
302             }
303         }
304         // calculate totals
305         var all Sizes
306         for _, v := range byMime {
307             all.TotalSize += v.RequestSize + v.ResponseSize
308             all.RequestSize += v.RequestSize
309             all.Count += v.Count
310             all.ResponseSize += v.ResponseSize
311             all.ResponseLines += v.ResponseLines
312             all.ExpandedResponseLines +=
    v.ExpandedResponseLines
313             all.ExpandedResponseSize += v.ExpandedResponseSize
314
315         }
316         byMime["*/*"] = all
317
318         keys := slices.Collect(maps.Keys(byMime))
```

```
319         slices.SortFunc(keys, func(a, b string) int {
320             return byMime[a].TotalSize - byMime[b].TotalSize
321         })
322
323         return Summary{
324             Title:          a.Log.Pages[0].Title,
325             ContentLoad:
      time.Duration(a.Log.Pages[0].PageTimings.OnContentLoad) *
      time.Millisecond,
326             Load:
      time.Duration(a.Log.Pages[0].PageTimings.OnLoad) *
      time.Millisecond,
327             TotalRequests: len(a.Log.Entries),
328             TotalSize:     all.TotalSize,
329             ByMime:        byMime,
330             Requests:      requests,
331         }, nil
332
333     }
```

anhar results

```
1   #!/usr/bin/env bash
2   go run ./anhar dumps/**/* | uniq
    # suppress repeated runs of table separators
```

| URL | MIME | Count | Request Size | Response Size (minified) | Response Size (expanded) | Response Lines | Response Lines (expanded) | Content-Load | Load

codeberg.org/efronlicht/ghsucks	image/png	1	963 B	5 KiB	5 KiB	1	1	2934 ms	3172 ms
codeberg.org/efronlicht/ghsucks	image/svg+xml	3	2 KiB	11 KiB	12 KiB	181	257	2934 ms	3172 ms
codeberg.org/efronlicht/ghsucks	text/html; charset=utf-8	1	1011 B	60 KiB	258 KiB	1099	2491	2934 ms	3172 ms
codeberg.org/efronlicht/ghsucks	text/css; charset=utf-8	3	2 KiB	434 KiB	434 KiB	6	27	2934 ms	3172 ms
codeberg.org/efronlicht/ghsucks	text/javascript; charset=utf-8	3	2 KiB	1 MiB	1 MiB	116	704	2934 ms	3172 ms
codeberg.org/efronlicht/ghsucks	*/*	11	9 KiB	1 MiB	1 MiB	1403	3480	2934 ms	3172 ms
—	—	—	—	—	—	—	—	—	—
codeberg.org/efronlicht/ghsucks/issues	image/png	1	963 B	5 KiB	5 KiB	1	1	3394 ms	3412 ms
	image/svg+xml	3				181	257		

codeberg.org/efronlicht/ghsucks/issues			2 KiB	11 KiB	12 KiB			3394 ms	3412 ms
codeberg.org/efronlicht/ghsucks/issues	text/html; charset=utf-8	1	1011 B	55 KiB	369 B	1109	1	3394 ms	3412 ms
codeberg.org/efronlicht/ghsucks/issues	text/css; charset=utf-8	2	1 KiB	433 KiB	433 KiB	4	4	3394 ms	3412 ms
codeberg.org/efronlicht/ghsucks/issues	text/javascript; charset=utf-8	2	1 KiB	1 MiB	1 MiB	114	114	3394 ms	3412 ms
codeberg.org/efronlicht/ghsucks/issues	*/*	9	7 KiB	1 MiB	1 MiB	1409	377	3394 ms	3412 ms
—	—	—	—	—	—	—	—	—	—
codeberg.org/efronlicht/ghsucks/pulls	image/png	1	963 B	5 KiB	5 KiB	1	1	2682 ms	2697 ms
codeberg.org/efronlicht/ghsucks/pulls	image/svg+xml	5	3 KiB	22 KiB	23 KiB	347	509	2682 ms	2697 ms
codeberg.org/efronlicht/ghsucks/pulls	text/html; charset=utf-8	1	1011 B	56 KiB	143 KiB	1112	1241	2682 ms	2697 ms
codeberg.org/efronlicht/ghsucks/pulls	text/css; charset=utf-8	2	1 KiB	433 KiB	433 KiB	4	4	2682 ms	2697 ms
codeberg.org/efronlicht/ghsucks/pulls	text/javascript; charset=utf-8	2	1 KiB	1 MiB	1 MiB	114	114	2682 ms	2697 ms
codeberg.org/efronlicht/ghsucks/pulls	*/*	11	9 KiB	1 MiB	1 MiB	1578	1869	2682 ms	2697 ms
—	—	—	—	—	—	—	—	—	—
codeberg.org/efronlicht/ghsucks/settings	image/png	1	963 B	5 KiB	5 KiB	1	1	2857 ms	-32 ms
codeberg.org/efronlicht/ghsucks/settings	image/svg+xml	6	3 KiB	23 KiB	23 KiB	348	510	2857 ms	-32 ms

codeberg.org/efronlicht/ghsucks/settings	text/html; charset=utf-8	1	1011 B	52 KiB	288 KiB	1149	3425	2857 ms	-32 ms
codeberg.org/efronlicht/ghsucks/settings	text/css; charset=utf-8	2	1 KiB	433 KiB	433 KiB	4	4	2857 ms	-32 ms
codeberg.org/efronlicht/ghsucks/settings	text/javascript; charset=utf-8	2	1 KiB	1 MiB	1 MiB	114	114	2857 ms	-32 ms
codeberg.org/efronlicht/ghsucks/settings	*/*	12	9 KiB	1 MiB	1 MiB	1616	4054	2857 ms	-32 ms
—	—	—	—	—	—	—	—	—	—
eblog.fly.dev/index.html	text/css; charset=utf-8	1	438 B	4 KiB	4 KiB	253	252	67 ms	200 ms
eblog.fly.dev/index.html	image/png	1	489 B	4 KiB	4 KiB	1	1	67 ms	200 ms
eblog.fly.dev/index.html	text/html; charset=utf-8	1	490 B	5 KiB	5 KiB	148	148	67 ms	200 ms
eblog.fly.dev/index.html	font/woff2	2	894 B	181 KiB	181 KiB	2	2	67 ms	200 ms
eblog.fly.dev/index.html	*/*	5	2 KiB	196 KiB	196 KiB	404	403	67 ms	200 ms
—	—	—	—	—	—	—	—	—	—
eblog.fly.dev/startingsystems3.html	text/css; charset=utf-8	1	449 B	4 KiB	4 KiB	253	252	76 ms	1109 ms
eblog.fly.dev/startingsystems3.html	image/png	1	500 B	4 KiB	4 KiB	1	1	76 ms	1109 ms
eblog.fly.dev/startingsystems3.html	font/woff2	2	894 B	181 KiB	181 KiB	2	2	76 ms	1109 ms
eblog.fly.dev/startingsystems3.html	text/html; charset=utf-8	1	520 B	576 KiB	576 KiB	3619	3619	76 ms	1109 ms
eblog.fly.dev/startingsystems3.html	*/*	5	2 KiB	766 KiB	766 KiB	3875	3874	76 ms	1109 ms
github.com/ef0xa/ghsucks	image/svg+xml	2	1020 B	3 KiB	3 KiB	8	30	843 ms	21330 ms
github.com/ef0xa/ghsucks	application/x-unknown-content-type	1	1 KiB	4 KiB	4 KiB	1	1	843 ms	21330 ms
github.com/ef0xa/ghsucks	text/plain	12	34 KiB	7 KiB	7 KiB	12	12	843 ms	21330 ms
github.com/ef0xa/ghsucks	image/png	5	3 KiB	77 KiB	77 KiB	5	5	843 ms	21330 ms

github.com/ef0xa/ghsucks	application/json; charset=utf-8	11	20 KiB	75 KiB	76 KiB	13	100	843 ms	21330 ms
github.com/ef0xa/ghsucks	text/html; charset=utf-8	4	7 KiB	281 KiB	281 KiB	2551	2551	843 ms	21330 ms
github.com/ef0xa/ghsucks	text/css	42	18 KiB	2 MiB	2 MiB	85	60016	843 ms	21330 ms
github.com/ef0xa/ghsucks	application/javascript	214	89 KiB	12 MiB	19 MiB	1554	481849	843 ms	21330 ms
github.com/ef0xa/ghsucks	*/*	291	178 KiB	15 MiB	22 MiB	4229	544564	843 ms	21330 ms
—	—	—	—	—	—	—	—	—	—
github.com/ef0xa/ghsucks/pulls	image/svg+xml	2	1020 B	3 KiB	3 KiB	8	30	592 ms	6754 ms
github.com/ef0xa/ghsucks/pulls	text/plain	11	29 KiB	7 KiB	7 KiB	11	11	592 ms	6754 ms
github.com/ef0xa/ghsucks/pulls	image/png	2	2 KiB	53 KiB	53 KiB	2	2	592 ms	6754 ms
github.com/ef0xa/ghsucks/pulls	application/json; charset=utf-8	7	12 KiB	53 KiB	53 KiB	9	96	592 ms	6754 ms
github.com/ef0xa/ghsucks/pulls	text/html; charset=utf-8	2	3 KiB	154 KiB	154 KiB	1246	1246	592 ms	6754 ms
github.com/ef0xa/ghsucks/pulls	text/css	40	17 KiB	1 MiB	2 MiB	81	56172	592 ms	6754 ms
github.com/ef0xa/ghsucks/pulls	application/javascript	202	84 KiB	11 MiB	18 MiB	1526	430365	592 ms	6754 ms
github.com/ef0xa/ghsucks/pulls	*/*	266	151 KiB	14 MiB	20 MiB	2883	487922	592 ms	6754 ms
—	—	—	—	—	—	—	—	—	—
github.com/ef0xa/ghsucks/settings	image/svg+xml	2	1020 B	3 KiB	3 KiB	8	30	778 ms	6963 ms
github.com/ef0xa/ghsucks/settings	text/plain	10	27 KiB	6 KiB	6 KiB	10	10	778 ms	6963 ms
github.com/ef0xa/ghsucks/settings	image/png	2	2 KiB	53 KiB	53 KiB	2	2	778 ms	6963 ms
github.com/ef0xa/ghsucks/settings	application/json; charset=utf-8	7	12 KiB	53 KiB	54 KiB	9	96	778 ms	6963 ms
github.com/ef0xa/ghsucks/settings	text/html; charset=utf-8	1	1 KiB	293 KiB	293 KiB	3425	3425	778 ms	6963 ms

github.com/ef0xa/ghsucks/settings	text/css	39	17 KiB	1 MiB	2 MiB	79	55035	778 ms	6963 ms
github.com/ef0xa/ghsucks/settings	application/javascript	194	81 KiB	11 MiB	17 MiB	1489	427469	778 ms	6963 ms
github.com/ef0xa/ghsucks/settings	*/*	255	144 KiB	14 MiB	20 MiB	5022	486067	778 ms	6963 ms
gitlab.com/efronlicht/ghsucks/-/boards	application/x-unknown-content-type	1	548 B	279 B	279 B	1	1	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	text/plain;charset=UTF-8	1	3 KiB	311 B	309 B	1	1	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	application/json	2	1 KiB	2 KiB	2 KiB	2	2	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	image/png	3	7 KiB	37 KiB	16 KiB	3	3	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	text/html;charset=utf-8	1	2 KiB	52 KiB	52 KiB	283	283	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	application/json;charset=utf-8	9	40 KiB	46 KiB	30 KiB	9	9	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	image/svg+xml	1	2 KiB	215 KiB	233 KiB	1	4012	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	application/octet-stream	4	9 KiB	1 MiB	1 MiB	4	4	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	text/css	9	21 KiB	1 MiB	1 MiB	9336	12626	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	text/javascript	26	62 KiB	4 MiB	2 MiB	141	41722	6621 ms	11822 ms
gitlab.com/efronlicht/ghsucks/-/boards	*/*	57	151 KiB	7 MiB	5 MiB	9781	58663	6621 ms	11822 ms
—	—	—	—	—	—	—	—	—	—
gitlab.com/efronlicht/ghsucks	application/x-unknown-content-type	1	548 B	279 B	279 B	1	1	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	text/plain;charset=UTF-8	1	3 KiB	311 B	309 B	1	1	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	application/json	3	6 KiB	4 KiB	4 KiB	3	3	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	image/png	4	9 KiB	47 KiB	26 KiB	4	4	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	text/html;charset=utf-8	1	2 KiB	65 KiB	260 KiB	455	2491	6880 ms	12941 ms
		10				10	10		

gitlab.com/efronlicht/ghsucks	application/json; charset=utf-8		34 KiB	37 KiB	35 KiB			6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	image/svg+xml	3	7 KiB	433 KiB	467 KiB	3	6673	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	application/octet-stream	4	9 KiB	1 MiB	1 MiB	4	4	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	text/css	10	24 KiB	1 MiB	1 MiB	9338	13643	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	text/javascript	41	97 KiB	5 MiB	4 MiB	212	78852	6880 ms	12941 ms
gitlab.com/efronlicht/ghsucks	*/*	78	196 KiB	8 MiB	7 MiB	10031	101682	6880 ms	12941 ms
—	—	—	—	—	—	—	—	—	—
gitlab.com/efronlicht/ghsucks/-/merge_requests	“	1	403 B	0 B	0 B	1	1	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	application/x-unknown-content-type	2	1 KiB	558 B	558 B	2	2	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	application/json	2	1 KiB	2 KiB	2 KiB	2	2	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	application/json; charset=utf-8	2	7 KiB	7 KiB	6 KiB	2	2	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	text/plain; charset=UTF-8	2	15 KiB	622 B	618 B	2	2	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	text/html; charset=utf-8	1	2 KiB	56 KiB	56 KiB	370	370	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	image/png	4	7 KiB	52 KiB	16 KiB	4	4	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	image/svg+xml	2	4 KiB	217 KiB	235 KiB	2	4054	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	application/octet-stream	4	9 KiB	1 MiB	1 MiB	4	4	6947 ms	12096 ms
	text/css	10	24 KiB	1 MiB	1 MiB	9338	13043	6947 ms	12096 ms

gitlab.com/efronlicht/ghsucks/-/merge_requests									
gitlab.com/efronlicht/ghsucks/-/merge_requests	text/javascript	35	84 KiB	4 MiB	3 MiB	158	73083	6947 ms	12096 ms
gitlab.com/efronlicht/ghsucks/-/merge_requests	*/*	65	159 KiB	7 MiB	6 MiB	9885	90567	6947 ms	12096 ms
—	—	—	—	—	—	—	—	—	—
gitlab.com/efronlicht/ghsucks/edit	„	1	403 B	0 B	0 B	1	1	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	application/x-unknown-content-type	2	1 KiB	558 B	558 B	2	2	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	application/json	3	4 KiB	3 KiB	3 KiB	3	3	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	text/plain; charset=UTF-8	2	15 KiB	622 B	618 B	2	2	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	application/json; charset=utf-8	4	12 KiB	27 KiB	13 KiB	4	4	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	text/html; charset=utf-8	1	2 KiB	72 KiB	72 KiB	668	668	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	image/svg+xml	3	2 KiB	216 KiB	233 KiB	3	4014	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	image/png	57	140 KiB	465 KiB	444 KiB	57	57	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	application/octet-stream	4	9 KiB	1 MiB	1 MiB	4	4	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	text/css	10	24 KiB	1 MiB	1 MiB	9338	13011	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	text/javascript	30	71 KiB	4 MiB	3 MiB	142	54150	6964 ms	13302 ms
gitlab.com/efronlicht/ghsucks/edit	*/*	117	285 KiB	7 MiB	6 MiB	10224	71916	6964 ms	13302 ms
—	—	—	—	—	—	—	—	—	—

bonus content: `testcompress`

testcompress source

```go
// testcompress tests whether a URL supports gzip and zstd compression and prints the results in a markdown table.
// Usage: testcompress path/to/url [path/to/url2]...
// If it does NOT support compression, it will compress the response body to show the potential savings. If it does support compression, it will decompress the response body to show the uncompressed size.
package main

import (
    "compress/gzip"
    "context"
    "fmt"
    "io"
    "log"
    "net/http"
    "os"
    "strings"
    "sync"
    "text/tabwriter"

    "github.com/klauspost/compress/zstd"
)

func main() {

    if len(os.Args) < 2 {
        log.Fatal("Usage: testcompression path/to/url [path/to/url2]...")
    }
    const padding, minwidth, tabwidth, padchar, flags = 2, 4, 4, ' ', 0
    tw := tabwriter.NewWriter(os.Stdout, padding, minwidth, tabwidth, padchar, flags)
    defer tw.Flush()

    const format = "|%v|%v|%v|%v|%v|%v|%v|%v|\n"
    fmt.Fprintf(tw, format, "URL", "MIME", "supported encodings", "Bytes", "Bytes (gzip)", "Bytes (zstd)", "Ratio (gzip)", "Ratio (zstd)")
    fmt.Fprintf(tw, format, "----", "----", "----", "----", "----", "----", "----", "----")
    urls := os.Args[1:]

    wg := new(sync.WaitGroup)
    for _, url := range urls {
        wg.Go(func() {
            req, err := http.NewRequest("GET", url, nil)
            if err != nil {
                panic(err)
            }
            req.Clone(context.TODO())
            req.Header.Set("Accept-Encoding", "") // grab it without any compression to see the uncompressed size
```

```
44                resp, err := http.DefaultClient.Do(req)
45                if err != nil {
46                    log.Printf("Error fetching %s: %v", url,
    err)
47                    return
48                }
49                if resp.StatusCode != http.StatusOK {
50
    log.Printf("Unexpected status code for %s: %d", url,
    resp.StatusCode)
51                    return
52                }
53                body, err := io.ReadAll(resp.Body)
54                if err != nil {
55                    log.Printf("Error reading response body
    for %s: %v", url, err)
56                }
57                if err := resp.Body.Close(); err != nil {
58                    log.Printf("Error closing response body
    for %s: %v", url, err)
59                }
60                zstdLen := count(writeZstd, body)
61                gzipLen := count(writeGzip, body)
62                zstdRatio := float64(zstdLen) /
    float64(len(body)) * 100
63                gzipRatio := float64(gzipLen) /
    float64(len(body)) * 100
64
65                var supportString string
66                {
67                    if supports(url, "zstd") {
68                        supportString = "zstd"
69                    }
70                    if supports(url, "gzip") {
71                        supportString += " gzip"
72                    }
73                    supportString =
    strings.TrimSpace(supportString)
74                }
75                fmt.Fprintf(tw, format,
76                    fmt.Sprintf("[%s](%s)",
    strings.TrimPrefix(url, "https://"), url),
77                    resp.Header.Get("Content-Type"),
78                    supportString,
79                    formatBytes(int64(len(body))),
80                    formatBytes(gzipLen),
81                    formatBytes(zstdLen),
82                    fmt.Sprintf("%.2f%%", gzipRatio),
83                    fmt.Sprintf("%.2f%%", zstdRatio),
84                )
85            })
86        }
87        wg.Wait()
88
```

```
 89  }
 90  func writeZstd(w io.Writer) io.WriteCloser {
 91      zw, _ := zstd.NewWriter(w,
     zstd.WithEncoderLevel(zstd.EncoderLevelFromZstd(19)))
 92      return zw
 93  }
 94  func writeGzip(w io.Writer) io.WriteCloser {
 95      gw, _ := gzip.NewWriterLevel(w, gzip.BestCompression)
 96      return gw
 97  }
 98
 99  func count(f func(w io.Writer) io.WriteCloser, b []byte)
     int64 {
100      var count countingWriter
101      w := f(&count)
102      if _, err := w.Write(b); err != nil {
103          return -1
104      }
105      if err := w.Close(); err != nil {
106          return -1
107      }
108      return int64(count)
109  }
110
111  func supports(url, encoding string) bool {
112      req, err := http.NewRequest("GET", url, nil)
113      if err != nil {
114          panic(err)
115      }
116      req.Header.Set("Accept-Encoding", encoding)
117      resp, err := http.DefaultClient.Do(req)
118      if err != nil {
119          log.Printf("check %s for %s support: %v", url,
     encoding, err)
120          return false
121      }
122      if resp.StatusCode != http.StatusOK {
123          log.Printf("check %s for %s support: unexpected
     status code: %d", url, encoding, resp.StatusCode)
124          return false
125      }
126      return resp.Header.Get("Content-Encoding") == encoding
127  }
128  func formatBytes(b int64) string {
129      if b < 1024 {
130          return fmt.Sprintf("%d B", b)
131      }
132      if b < 1024*1024 {
133          return fmt.Sprintf("%.2f KiB", float64(b)/1024)
134      }
135      return fmt.Sprintf("%.2f MiB", float64(b)/1024/1024)
136
137  }
138
```

```
139  // countingWriter is an io.Writer that counts the number
     of bytes written to it.
140  type countingWriter int
141
142  // "write" bytes to the countingWriter by incrementing the
     count by the length of the byte slice.
143  func (cw *countingWriter) Write(p []byte) (int, error) {
144      *cw += countingWriter(len(p))
145      return len(p), nil
146  }
147
```

bonus content: `testcompress` results:

```
1  #!/usr/bin/env bash
2  go run ./testcompress \
3      'https://eblog.fly.dev/startingsystems3.html' \
4      'https://github.com/ef0xa/ghsucks' \
5      'https://gitlab.com/efronlicht/ghsucks' \
6      'https://codeberg.org/efronlicht/ghsucks'
```

URL	MIME	supported encodings	Bytes	Bytes (uncompressed)	Bytes (zstd)	Ratio (gzip)	Ratio (zstd)
eblog.fly.dev/startingsystems3.html	text/html; charset=utf-8	zstd gzip	575.77 KiB	67.64 KiB	60.17 KiB	11.75%	10.45%
github.com/ef0xa/ghsucks	text/html; charset=utf-8	gzip	224.70 KiB	43.27 KiB	37.96 KiB	19.26%	16.90%
gitlab.com/efronlicht/ghsucks	text/html; charset=utf-8	gzip	43.08 KiB	11.48 KiB	11.34 KiB	26.64%	26.32%
codeberg.org/efronlicht/ghsucks	text/html; charset=utf-8	n/a	43.88 KiB	13.00 KiB	12.79 KiB	29.62%	29.16%

MORE ARTICLES