Episode Book

EPISODE #5942

Service Workers: The Proxy That Never Sleeps

PUBLISHED	14 October 2026 at 00:18 UTC
DURATION	27:57
LISTEN	myweirdprompts.com/episode/service-worker-lifecycle-explained-19120/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.4
SOURCES	14 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:41 UTC

SYNOPSIS

Service workers sit between your page and the network as a programmable proxy, running on their own thread with no DOM access and no window. They can wake up when no tab is open, intercept every request under their scope, and serve content from a cache you control — which is exactly why a deploy can appear to do nothing at all. This episode walks through the full lifecycle: download, install, activate, and the "waiting" state that explains why refreshing a page never picks up a new version. Along the way: why AppCache had to die, what clients.claim and skipWaiting actually trade away, the 24-hour update check window, and the clone-the-stream footgun that turns a five-minute mistake into a four-hour debugging session. We also cover cache-first vs network-first vs stale-while-revalidate, navigation preload, Background Sync, and how push notifications wake a worker on a device that has no tab open.

TRANSCRIPT



Corn

There's a specific kind of developer despair that only shows up in one place. You deploy a fix, you reload the page, and the site is exactly as broken as it was five minutes ago. Nothing changed. You check the network tab. The response came back instantly.



Herman

From cache.



Corn

And the cache isn't the browser's, and it isn't the CDN's.



Herman

It's the service worker's.



Corn

Daniel wants to know what that thing actually is. He's been poking at it while debugging, and his question is basically: why does this exist at all when we already have JavaScript running in the page? How does a file run with no tab open, intercept every request the site makes, and then, as he puts it, serve outdated content even after a refresh? And then the bigger one underneath it, which is whether this is the future of the web or a specialized trick that most people have installed without realizing it.



Herman

That last part is the one with the interesting answer.



Corn

So. What is it.



Herman

A service worker is a JavaScript file registered against an origin. The browser downloads it, installs it, and then keeps it around, on its own thread, with no DOM access, no window, no localStorage. What it has instead is the ability to sit between the page and the network as a programmable proxy. Every request the site makes under its scope passes through it, and the worker gets to decide what comes back.



Corn

And the reason the page's own JavaScript can't do that job?



Herman

A page script only exists while the page exists. It can't see the browser's own navigation request. It can't run when the tab is closed. Nothing can wake it up from outside the tab. A service worker can do all three of those things. MDN's framing is that before service workers there just wasn't a good overall control mechanism for asset caching and custom network requests. You had the browser cache, which you couldn't program, and you had Application Cache.



Corn

AppCache.



Herman

AppCache is the reason service workers exist in the shape they do. It was simple. It was also rigid, and it broke in ways nobody could see coming. MDN's line on it is that AppCache, quote, makes assumptions about what you are trying to do, and then breaks when those assumptions are not exactly right.



Corn

That's a haunted-house description of a caching layer.



Herman

Jake Archibald wrote in 2014 that AppCache made this mistake and everybody's left dealing with the consequences. Which is a fair summary of why the replacement is so deliberately paranoid. The service worker spec's first public working draft landed in May 2014. Chrome shipped it in beta that December. Everything in the design comes out of that post-mortem.



Corn

Okay, so here's the tension I think the whole episode lives in. The thing that makes a service worker safe is the same thing that makes it maddening.



Herman

The lifecycle is where that shows up.



Corn

Go.



Herman

You register with ServiceWorkerContainer.register, pointing at a script. Then there are three phases. Download, install, activate.



Corn

Install is where you precache.



Herman

Install fires once. That's your moment to pull the core assets into the Cache Storage API, your shell HTML, your CSS, your main bundle. Activate is where you clean up, delete caches from previous versions. Between those two events, the worker is live but not yet controlling anything.



Corn

And then the part everyone gets wrong on the first day.



Herman

On the very first visit, the page's own requests do not go through the worker. Not unless the page request itself came through it, which on a first load it didn't. So you register your worker, you feel good about it, you reload, and nothing is different. web.dev's line is that you'll need to refresh the page to see the effects of the service worker. That's the first-load paradox. The worker is installed but idle as far as that page is concerned.



Corn

And there's an API that breaks that.



Herman

clients.claim. Call it inside activate and the worker takes control of the pages that are already open. It's convenient and it's a little bit rude. You're adopting clients that never agreed to be managed by you.



Corn

Fits the theme.



Herman

Then the interesting part, which is updates. You push a new version of the worker script. The browser downloads it and installs it. And then it sits there. It goes into a state literally called waiting.



Corn

Waiting for what.



Herman

For no pages to be controlled by the old worker.



Corn

So you close all your tabs.



Herman

You close all your tabs. But here's the subtlety that eats entire afternoons. Refreshing is not enough. web.dev's line is that the current service worker is always controlling a client during a refresh. You hit reload, the request goes out, and the old worker answers it. The worker is still there, still controlling, so the new one keeps waiting. You have to actually close every tab on the origin. Not just the one. Every one.



Corn

So the workflow is: deploy, watch nothing happen, remember how service workers work, close eleven tabs you were using for other things, reopen, and now it updates.



Herman

That's the honest workflow. And there's an escape hatch. self.skipWaiting. Call it and the new worker skips the wait and takes over immediately.



Corn

Which sounds like the obvious thing to do and is apparently a trap.



Herman

It's a tradeoff, not a trap. The waiting state exists for a reason. If two versions of your app are live in different tabs at the same time, you can get real problems, one tab posting data in a shape the other one no longer understands. The conservatism is protecting you. skipWaiting throws that away to make your deploy feel fast.



Corn

Right, so if I'm building a chat app, skipWaiting is fine. If I'm building something with a form and a session, skipWaiting can lose somebody's work.



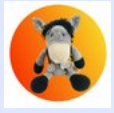
Herman

Which is exactly why Slack's team wrote a whole post about accepting being one version behind.



Corn

Hold that. The update checks. When does the browser ever go looking for a new worker?



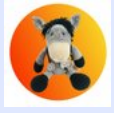
Herman

On every in-scope navigation, and on functional events, so push, sync. With one exception. If there's been a check in the last twenty-four hours, it skips.



Corn

Eighty-six thousand four hundred seconds.



Herman

That number is worth knowing because it explains a class of bug that looks like time travel. Somebody pushes a fix, a user navigates around, and the old version hangs on for a day. And since Chrome 68, browsers stopped honoring HTTP caching headers when checking the worker script itself. So you can't cache-control your way out of an update check.



Corn

Good. What about when the worker isn't doing anything.



Herman

It isn't running. MDN says browsers may stop service workers when they think it's appropriate, and restart them when an event arrives that they need to handle. In Chrome the numbers are roughly thirty seconds idle, thirty seconds of synchronous JavaScript, or a waitUntil promise that runs over five minutes. It's a dormant thing. It wakes up, does a job, goes back to sleep.



Corn

That's a very appealing architecture and I want to say so carefully.



Herman

Say it carefully.



Corn

A process that does nothing until something actually needs it, and then handles it and goes dormant. I approve of that and I'd like it noted that I approve of it for reasons unrelated to my own habits.



Herman

Noted. Nobody's going to check that.



Corn

The interception. How does the worker actually see a request.



Herman

Every request under the worker's scope fires a fetch event. The worker gets the event, and if it wants to answer, it calls event.respondWith and hands back a promise for a response. That response can be anything. It can come from Cache Storage. It can come from the network. It can be something you construct in the worker itself.



Corn

And matching against the cache is by URL and headers.



Herman

caches.match on the request object. URL and headers, not the body.



Corn

So two POSTs to the same endpoint look identical to the cache.



Herman

Which is why you don't cache POSTs. And there's a genuine footgun in cloning. A request and a response stream can only be read once. If you want to return the response to the page and also store it, you have to clone it first. Use the original, cache the clone, or the reverse. Skip that and you get a body that's already been consumed.



Corn

And once it's consumed it's gone.



Herman

It's gone. You can't re-read a stream. That's a five-minute mistake and a four-hour debugging session, because the error doesn't say "you needed to clone this." It says something about a disturbed stream.



Corn

The strategies. There are basically three.



Herman

Cache-first, network-first, and stale-while-revalidate. Cache-first checks Cache Storage, returns if there's a hit, falls back to the network if not. It's fast, and MDN is blunt that cache-first strategies are more likely to return a stale response. Network-first goes to the network, falls back to the cache if the network fails, so it's fresh when online and degraded-but-working offline. Stale-while-revalidate is the clever one. It returns the cached copy immediately and updates the cache in the background for next time.



Corn

So the user gets yesterday's version today, and tomorrow they get today's.



Herman

And most users never notice. That's the point.



Corn

There's a thing for the startup delay on navigation.



Herman

Navigation preload. Normally, on a navigation, the browser has to start the worker, and only then can the worker make a network request, so you've added worker startup time to your page load. Navigation preload starts that network request in parallel with worker startup, so the request isn't waiting on the worker. It's on about two percent of service workers, which we'll come back to.



Corn

What does all this actually buy you. Offline, sync, push.



Herman

Offline is the cache strategies doing their job. Sync is the interesting one. Without it, a user hits send on a form, they're on a train, there's no signal, and Jake Archibald's framing is that they have to stare at a spinner until it completes. Which, if the connection never comes back, is forever.



Corn

So they close the tab and the message is gone.



Herman

Background Sync shipped in Chrome 49, in December 2015. The app registers a named sync when the user sends. If the network is down, the send is queued. When connectivity comes back, the browser restarts the worker and fires a sync event, and the worker replays the request.



Corn

The tab might not even be open.



Herman

Might not be open. The worker handles it independently.



Corn

Push.



Herman

Your server sends an encrypted, signed message to a push service, which the browser vendor operates. Chrome's is Google's, Firefox's is Mozilla's. That message gets delivered to the browser, the browser starts the worker, fires a push event, and the worker calls registration.showNotification.



Corn

So a server can wake up a worker on a device.



Herman

That's the design. Silent push is allowed by the spec and no browser implements it, for privacy reasons. Every push has to result in something the user can see. If you want to be cynical you can say the spec wrote down the constraint and the vendors enforced it.



Corn

I'd rather not be cynical about that one.



Herman

There's also Background Fetch and Periodic Background Sync. Background Fetch handles a long download, a movie or a large file, with a persistent browser UI the user can cancel. Periodic Background Sync refreshes data on an interval that the browser itself decides, based on how often the user actually engages with your app.



Corn

Which is a nice inversion. The app doesn't set the interval.



Herman

The browser does, and it's tied to engagement. If a user opens your app twice a week, you get refreshed twice a week. If they never open it, you don't get to spend their battery.



Corn

So the mechanism is conservative end to end. And that conservatism is precisely what makes this thing so miserable to debug. Let's do the stale content question properly, because it's the one Daniel asked outright.



Herman

The classic case. You deploy a new index.html and a new worker. The user's browser has the old worker. The old worker's cache-first fetch handler answers the navigation request. And it answers it with the old index.html, which it has cached, because that's what cache-first means.



Corn

So they get the old page.



Herman

They get the old page. And the old page registers the old worker URL, which is still in the cache and still installed, so the whole system is self-consistent. It's internally correct and externally wrong. web.dev's line on this is that you've put yourself in a position where you need to update your service worker in order to update your service worker.



Corn

Ew.



Herman

That's their word, and they earned it. And there's a worse variant. Say you versioned the worker URL. Your old worker was at v3 dot js, your new one is at v4 dot js. You deploy v4 and you delete v3. Now the browser goes to check for an update on v3, gets a 404, and the old worker stays active indefinitely. It doesn't know v3 is gone. It just knows it can't update.



Corn

So it keeps serving the old site forever.



Herman

Until you either close every tab on that origin, or ship something that unregisters it.



Corn

And reloading doesn't help.



Herman

Reloading requests the document again, which the service worker answers from the same cache, so you get the same broken document. That's a direct quote from a writeup by a developer who hit exactly this in production. And then the discovery most people make too late: shift-reload bypasses the service worker entirely. That's spec'd behavior. A hard reload goes straight to the network.



Corn

Hold on. That's the escape hatch, and it reads to me like the single most useful sentence anyone could put in front of a developer hitting this for the first time.



Herman

It's the sentence that ends the afternoon. And almost nobody knows it, because a normal reload and a hard reload look identical if you don't know the worker is in the middle.



Corn

What else pins stale or wrong content.



Herman

cache.put accepts error responses. That's the API being honest about what it is. Only add and addAll reject non-OK responses. So if you use put and the response was a 502, or worse, a captive portal login page from a hotel network, you've just cached that.



Corn

So the user connects to hotel wifi, the site comes up as a login page, and that login page is now the site until they clear storage.



Herman

That's not a hypothetical pattern. And there's the opaque response problem. When you fetch a cross-origin resource without CORS, you get an opaque response. You can store it, you can serve it back, but you cannot inspect it at all. You can't tell if it's a 200 or a 500 or a picture of a cat. And Chromium charges those against your storage quota with padding, roughly seven megabytes each on average.



Corn

Seven megabytes of quota for a file you can't even verify.



Herman

The range runs from zero to something like fourteen thousand kibibytes. It's not a small thing if you're caching a lot of cross-origin assets.



Corn

And there's the Vary header thing.



Herman

cache.match matches on URL and headers. If your responses have a Vary header, the match logic can silently fail against requests that should have hit, and you get a cache miss that looks like the cache is broken. It's matching correctly. You just didn't know it was matching on that.



Corn

And skipWaiting is in the list of traps.



Herman

Because the moment you skip, the new worker activates and typically deletes the old caches. If a tab has been open for three hours and is mid-way through loading an asset from a cache you just deleted, that load fails. You didn't break the page. You broke the environment the page was living in.



Corn

The tools, then. Because part of why this is confusing is that the browser can barely tell you what's going on.



Herman

Chrome and Edge put everything in the Application panel. Registration state, Cache Storage, which requests were answered by the worker. It's all there. Firefox splits it across about:debugging and the Storage panel. Safari shows you registrations and no cache viewer at all.



Corn

So on Safari you can see the problem and not the contents.



Herman

You can see that a worker exists and that it's not the one you deployed. You cannot look inside its cache to find out why.



Corn

Which means the debugging loop is close the tab, hard reload, check the Application panel, guess, repeat. Tested by me, and I'd like it on record that I would not have chosen to build it this way.



Herman

You're the wrong shape for this technology generally.



Corn

I really am.



Herman

There's a specific reason the restrictions exist, and they're worth listing because they explain most of the weird behavior. Service workers are HTTPS-only, with localhost exempt so you can develop. They're same-origin only, which means you cannot host your service worker on a CDN. It has to come from your own origin. Its scope is limited to the directory it's served from, unless you widen it deliberately with a Service-Worker-Allowed header.



Corn

So a worker at slash scripts slash sw dot js controls slash scripts slash and below, and nothing above it.



Herman

Right. And you can't serve one from a third-party domain and control your whole site, which kills a whole category of shortcuts that people try first.



Corn

And inside the worker itself, no DOM. No synchronous APIs. No dynamic import.



Herman

No document, no window, no localStorage, because localStorage is synchronous. Push requires explicit user permission and has to be visible. Background fetch and periodic sync require their own permissions.



Corn

Why HTTPS across the board.



Herman

Because a persistent, programmable, origin-scoped network proxy that lives on the user's machine and can intercept every request is a fundamentally dangerous object. Every restriction here is the browser deciding what it will let a script do to a user's traffic. And a service worker can rewrite responses, so without HTTPS, an attacker on the network can trivially inject a service worker and then own everything the user sees on that site, permanently.



Corn

The restrictions aren't caution. They're the whole security model.



Herman

They are. And that's also why the conservatism in the lifecycle is not the browser being lazy. It's a design that assumes the worker is going to outlive the page, outlive the deploy, and maybe outlive your patience.



Corn

Now the adoption question, because this is where I think the episode earns its keep. Daniel asked how widely these are used. What's the honest number.



Herman

About one in five web properties has a service worker. The 2025 Web Almanac's figures, top one thousand sites, around thirty percent desk and twenty-nine mobile. Top ten thousand, a bit over twenty-nine. All sites, roughly twenty percent desktop, twenty mobile.



Corn

And what was it in 2022.



Herman

About one point seven percent.



Corn

So it went up roughly tenfold in three years.



Herman

Which sounds like a technology taking over the web. Except the Almanac explicitly says a lot of the growth is due to Google Tag Manager enabling service workers.



Corn

Google Tag Manager is installing them.



Herman

Not in the sense of you turning it on. The way the product works has produced service worker registrations on sites whose developers never typed the words "service worker" in their lives.



Corn

So the headline number is measuring something adjacent to adoption. It's measuring presence.



Herman

Presence. And the event usage data backs that up hard. Across all those service workers, activate fires on ninety-six percent, install on sixty-four, fetch on twelve, push and notificationclick on seven.



Corn

Fetch fires on twelve percent. The one event that's the entire point of the technology.



Herman

Twelve percent. Which means the overwhelming majority of installed service workers have no fetch handler. They never intercept a request. They register, they install, they activate, and then they sit there doing nothing.



Corn

I want to be fair to them. Doing nothing is a legitimate contribution.



Herman

It's a very Corn thing to say.



Corn

It's an efficient use of resources and I stand by it.



Herman

The picture gets sharper. skipWaiting is called on about sixty-eight percent of desktop workers and sixty-three percent mobile, which tells you most people are using the escape hatch rather than the default. pushManager registrations, about seven percent. sync dot register, about two percent. navigationPreload dot enable, about two percent.



Corn

So background sync, the feature Jake Archibald wrote a whole post about, is on two percent of deployments.



Herman

And navigation preload, the fix for the startup delay on navigation, also two percent. So the two features most likely to make this technology feel good to a user are on almost nothing.



Corn

What are the real implementations worth naming.



Herman

Slack's is the best documented. They published it in 2019. The service worker cut boot time by more than fifty percent, with about a twenty-five percent improvement going warm versus cold. They described tens of millions of requests a day going through millions of installed workers. Boot went from about five seconds to under one second.



Corn

And they made a specific choice about staleness.



Herman

They cache assets keyed by deploy timestamp and they accept being one version behind. They also re-register on a jittered interval so that not every client refreshes at the same moment. It's a team deciding a slightly old app is better than a synchronized stampede.



Corn

Do they go offline.



Herman

They kept API responses in the cache, not just static assets, so feature flags stay consistent with whatever version the client is running. A one-version-behind client gets one-version-behind feature flags, which sounds wrong and is actually the right thing.



Corn

Anything outside the browser.



Herman

Cloudflare Workers, from 2017, borrowed the service worker model and moved it to the edge. Same programming shape, entirely different execution environment. And Mock Service Worker uses them for API mocking in tests, which is a clever repurposing. You install a service worker in a test environment and your test framework intercepts every network call without running a real server.



Corn

What's the LaTeX one.



Herman

A browser-based LaTeX and WebAssembly editor that uses a service worker to validate and cache downloads, so the big WebAssembly toolchain doesn't get re-downloaded every time.



Corn

Which is the pattern I'd expect from the whole category. Heavy assets, intermittent connectivity, and the worker becomes the thing that turns an unusable web app into a usable one.



Herman

And that's the tension for the last part of this. The installability rule has loosened. Chrome and Edge no longer require a service worker to show the install prompt.



Corn

A site can be installed without one.



Herman

Installed, yes, still with the manifest and HTTPS. And the service worker is still required for the offline experience to be good.



Corn

The technology is decoupling from the story it was sold with. The PWA pitch was manifest plus service worker plus HTTPS, all three, and now it's manifest plus HTTPS and the service worker is optional.



Herman

Optional for install. Not optional for offline.



Corn

Here's where I land on Daniel's bigger question. If twenty percent of sites have a service worker and two percent of those use background sync, the technology isn't being adopted. It's being installed. Those are different things, and the number measures the second one.



Herman

Which raises the question of whether that matters. You could argue that a technology which is present everywhere and used shallowly is just infrastructure. Nobody says the web adopted TCP because every site uses it.



Corn

That's the strongest argument against my position and I don't fully have an answer to it.



Herman

I'll take that.



Corn

But there's a difference between infrastructure that's transparent and infrastructure that changes behavior. A service worker that installs, activates, and does nothing is not transparent. It's a persistent object on the user's machine, tied to an origin, that the browser has to maintain and that the user can't see. It's occupying a slot.



Herman

That gets us to something I've been waiting to say. The feature and the bug are the same mechanism. The waiting state that protects your data is the thing that makes your deploy look like it didn't happen. The cache-first strategy that makes a page load in fifty milliseconds is the strategy that serves yesterday's HTML. The conservatism is not a rough edge on top of the design. It is the design.



Corn

A developer choosing a service worker isn't choosing a feature. They're choosing a set of promises about staleness, and then living with them.



Herman

Most of the people who have one didn't choose anything.



Hilbert

I've got four of them on my laptop right now and none of them are mine. Different origins. The one that matters is a council licensing portal in Camden, for a food hygiene certificate. My brother-in-law runs a fish van and he put the site on his phone, and the thing has been showing him last November's menu since March. He rang the council



Hilbert (continued)

about it. They told him to clear his cookies. He cleared his cookies, which uninstalled the worker, and the site worked for six days, and then it came back. Because the site re-registers it every time he visits. I went through it with him on a Sunday. The worker had a fetch handler and a cache-first strategy, and it was caching the HTML, so the portal was pinned to whatever version was current in March. The council's own IT contractor billed them eleven thousand pounds to rebuild the front end in June. It went live on the fourth. Nobody who had visited before June ever saw it.



Corn

How did he get the certificate.



Hilbert

He didn't. He wrote a letter to the licensing office and posted it, and got a reply in eleven days, and the reply said his application was already on the system. He's had the certificate since June. He just can't see it on the website. He printed the letter and keeps it in the van. He's got four copies of the letter now, one for each van and one for the flat. I've got one of them, actually, in the glovebox. The other three are his. Mine's the pink one.



Corn

Okay, so where does that leave us on whether service workers are actually becoming central to the web.



Herman

If the closing question is whether twenty percent adoption with two percent sync usage is real, I think the answer is that the number tells you nothing about intent and everything about distribution. A feature that gets onto a fifth of the web by default is different from a feature that a fifth of developers chose.



Corn

Which is the thing I'd want a developer to take away. You're not choosing a caching library. You're choosing to own a piece of the user's browser that outlives your deploy and outlives your tab.



Herman

It will keep working exactly as designed the whole time.



Corn

Thanks to our producer Hilbert Flumingtop, who has a fish van story for everything. If you want more of this, try episode nineteen, AI Images; episode thirty-five, The Privacy Gap; and episode eleven, How Does Fine Tuning Work Anyway. This has been My Weird Prompts.



Herman

Send us your own prompt on Telegram at t dot me slash MWP listener bot. We'll be back soon.



Corn

My weird prompts dot com. See you.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **MDN [primary]**
   Service Worker API
   https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

2. **MDN [primary]**
   Using Service Workers
   https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers

3. **MDN [primary]**
   Offline and background operation (PWA guide)
   https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation

4. **web.dev [primary]**
   The service worker lifecycle (Jake Archibald, last updated 2016-09-29)
   https://web.dev/articles/service-worker-lifecycle

5. **Chrome for Developers [primary]**
   Introducing Background Sync (Jake Archibald, 2015-12-08)
   https://developer.chrome.com/blog/background-sync

6. **HTTP Archive / Web Almanac [primary]**
   PWA chapter 2025 (published 2026-01-16, updated 2026-05-05)
   https://almanac.httparchive.org/en/2025/pwa

7. **W3C [primary]**
   Service Workers First Public Working Draft (2014-05-08)
   https://www.w3.org/TR/2014/WD-service-workers-20140508/

8. **Chromium Blog [primary]**
   Chrome 40 Beta: Powerful Offline and Lightspeed Loading with Service Workers (2014-12-04)
   https://blog.chromium.org/2014/12/chrome-40-beta-powerful-offline-and.html

9. **HTTP Archive / Web Almanac**
   PWA chapters 2019, 2022 (usage history), https://almanac.httparchive.org/en/2020/pwa
   https://almanac.httparchive.org/en/2021/pwa

10. **Slack Engineering**
    Service Workers at Slack (2019-08-29, updated 2020-06-18)
    https://slack.engineering/service-workers-at-slack-our-quest-for-faster-boot-times-and-offline-support-3492cf79c88

11. **progressivewebapps.com**
    Service Worker Pitfalls & Anti-Patterns (2026-09-25)
    https://progressivewebapps.com/service-workers/pitfalls/

12. **JakeArchibald.com**
    Service Worker first draft (2014-05-08)
    https://jakearchibald.com/2014/service-worker-first-draft/

13. **maxdoubin.com**
    Service Worker Serving a Stale index.html After a Deploy
    https://maxdoubin.com/blog/service-worker-served-a-stale-document

14. **Hacker News**
    discussion threads on service workers (Cloudflare Workers 2017; MSW 2.0 2023; Slack 2019; iOS Web Push 2023; stale-cache anecdotes 2026), https://news.ycombinator.com/item?id=20838264 ; https://news.ycombinator.com/item?id=47675953
    https://news.ycombinator.com/item?id=15364896