Turtle: A From-Scratch Browser Rendering Engine in Zig

Hao Jiang

Abstract—Three engines render the web, each the product of years of work by a large team. This paper reports that a fourth is reachable by one developer. We built Turtle, a from-scratch browser rendering engine in Zig, together with the C ABI and the AppKit shell that drive it. The engine's central design is the parked-tree history cache: rather than storing a document and re-laying it out on restore, Turtle parks the already-laid-out fragment tree, so a back navigation skips layout entirely. The safety argument is that parking frees nothing—only eviction does—which removes the ambiguity a failed background worker would otherwise introduce.

The engine's architecture and its proof of concept were written by hand. An agent loop was adopted later, for the conformance grind—tens of thousands of WPT CSS layout tests—and its verification stage caught defects that would otherwise have shipped, most of them defects in the evidence that the code worked rather than in the code. Every number in this paper is emitted from a committed harness's CSV output: layout of a 144,586-px document in 1467 ms, effectively instant history-cache restore on small and medium pages against a ceiling on very large ones that we report as a negative result, deterministic layout across runs, and a score of 100 on Acid 3. The limitations include a reproducible layout panic and leaking tests.

I. Introduction

Three engines render the web. Blink, Gecko, and WebKit together account for the overwhelming majority of page views, and each is the product of years of engineering by a large team. That concentration is usually discussed as a governance problem, but it also rests on a technical premise that is rarely tested: that a rendering engine is simply too large an artifact for anyone outside those teams to produce. This paper is an existence proof that a fourth is reachable by one developer, and a report of which architectural choices made it tractable.

We built Turtle, a from-scratch browser rendering engine in Zig[1], and the AppKit shell that drives it. The originality claim is scoped precisely. Everything in the rendering path—style, layout, text, and paint—lives in src/render/: a cascade spanning hundreds of CSS properties, a layout mode for each formatting context (block, inline, flex, grid, and table), Core Text shaping, and a Metal rasterizer. Around it sit the C ABI (src/capi/, the 54 functions of Section 2) and the Swift shell. What that path reaches is a score of 100 on Acid 3 and 425 on html5test (Section 6). These are the artifacts this paper claims as original. The DOM and JavaScript plumbing are integration work on existing components, and we make no claim of novelty for them.

The paper makes three contributions. First, the engine itself and the design decisions that kept it tractable for a single author: a fragment-tree layout model, a two-arena lifetime discipline, and a poll-based C ABI that lets a single-threaded engine cooperate with the AppKit run loop. Second, the parked-tree history cache, which parks the laid-out fragment tree so that restore skips layout entirely—the paper's central technical contribution, and the one whose safety argument (parking frees nothing) is the interesting half. Third, the agent-loop methodology and the boundary at which it was adopted: the design and the proof of concept were written by hand, and a self-authored harness took over the conformance grind, where the loop's verification caught defects that would otherwise have shipped—thirteen vacuous assertions, a policy-inheritance hole, and a leak checker that exited zero while leaking. The measurements in this paper were produced by a committed harness against a fixed engine revision, so every number traces to a recorded run.

Turtle is not a shipping browser, its conformance is partial, and Section 9 lists a reproducible layout panic, leaking tests, and a platform boundary that no reading of the results should soften. The existence proof is about reachability, not parity.

II. Architecture

The engine renders a document as a fragment tree. Layout consumes the DOM—itself a tree of nodes—and produces a parallel tree of Fragment nodes, each carrying geometry and a pointer to the style that governs it. The fragment tree is what paint walks, and it is what the history cache parks (Section 5). Keeping the fragment tree separate from the DOM means layout can be recomputed, discarded, or parked without disturbing the source document, and it is the unit of work the rest of the paper measures. Structuring the engine around a single layout output that paint and the cache both consume is a deliberate departure from the process-per-site and compositor-first decompositions of the major engines[2], [3].

A. Two arenas, one lifetime

Memory is arena-allocated, and the arenas are chosen to match how long their contents are useful rather than how the code happens to be structured. Three of them do the work. doc_arena holds the style tree and lives until the document is replaced. layout_arena holds exactly one

layout's fragment tree and is reset when layout re-runs, so a relayout frees the previous tree in constant time instead of walking it. A third, anim_scratch, takes the transient slice payloads an animation tick interpolates—transforms, shadows, calculated lengths—so a continuously animating page does not grow the persistent arena a frame at a time.

The hazard is not inside any one arena but across two of them, and it is the source of every documented crash in the engine. A fragment's style field points into doc_arena. But the string fields of a ComputedStyle—font families, URLs, and other text—are slices into the Page's frame arena, which has a different owner and a different lifetime. The two are freed independently, so a fragment can outlive its style's strings, or a document can be torn down while a live fragment tree still holds pointers into it. Every SIGSEGV and SIGBUS we have documented in the renderer is an instance of that one divergence: a pointer valid in one arena and dangling in the other.

What makes the divergence reachable at all is that paint runs off the main thread. A worker walks the fragment tree while the main thread is free to navigate, so retiring a page is not simply a matter of freeing in the right order—it is a handshake with a walk already in progress. The failure mode is worse than a use-after-free of one field: swapping the page under the worker hands it a different fragment tree, backed by a different doc_arena and a different Page's frame arena, in the middle of a walk that is reading the old one.

The engine's answer is PageEntry, a single object holding a page and everything laid out from it—the document arena, the style tree, the fragment tree, the shaper memo, the scroll offset baked into that tree, and the content height published to the host. These had been loose fields on the view object, each retired independently and therefore each retired in some order relative to the paint worker. Binding them into one object collapses that: there is one pointer to hand over and one function, setCurrent, that hands it, and that function performs the join with the paint worker itself and returns whether the worker actually stopped.

It would overstate the result to say the divergence is now unrepresentable. What the bundling buys is narrower: it concentrates the hazard. Four fields that each had to be retired in the right order became one pointer with one writer, so getting it wrong is possible in one place instead of four. Half of the remaining rule is enforced by the compiler and half is not, and the difference is a property of the language. Zig rejects an ignored non-void return, so a caller that ignores setCurrent's refusal does not compile—the case where the worker would not stop cannot be silently dropped. Zig has no field privacy, so assigning the pointer directly and bypassing the setter does compile; that half is carried by a documented invariant rather than by the type system. A design that concentrates a hazard and then says exactly which half of it the compiler holds is more useful than one that claims to have eliminated it.

B. A poll-based C ABI

The engine exposes itself to the shell through a C ABI of 54 functions and no callbacks. The shell never hands the engine a function to call when something changes; the engine is polled instead. That is a deliberate fit for a single-threaded engine under an AppKit run loop. AppKit owns the main thread and drives it with events, so a callback-based engine would have to re-enter itself from inside an AppKit callback—which means being safe against arriving at arbitrary points in the run loop, at arbitrary depths of an existing stack. A poll model inverts that: the engine runs to a quiescent state, returns control, and the shell asks again on the next run-loop turn. The engine never re-enters itself, the run loop never blocks on the engine, and neither owns the other's stack.

Two conventions carry the model. First, work is bounded rather than run to completion: cw_view_pump takes a millisecond budget and returns whether there is more to do, so a slow page yields the main thread back to AppKit instead of holding it. cw_view_render likewise returns whether it actually produced a frame, which is what lets the shell's display loop ask on every turn while the engine answers only when the backing store changed (Section 3).

Second, every event that a callback API would push, this one makes the shell pull. Requests that originate in the page and need the host to answer—the fullscreen, geolocation, and clipboard permissions and their results—are exposed as take-shaped functions: the shell drains a pending request on its own turn, and answers later through a separate entry point carrying the request's identifier. An asynchronous permission prompt therefore becomes two ordinary polls rather than a re-entrant callback, and the engine's state machine only ever advances on a call the shell chose to make. The cost is that the shell must poll for things it did not ask about; the benefit is that there is no point in the engine where arbitrary host code can run.

III. Paint

Text is shaped with Core Text. CoreTextShaper.zig turns a run of fragments into glyph runs, asking the system for font metrics, kerning, and ligatures rather than reimplementing any of it. The engine does not rasterize glyphs itself; it hands Core Text's output to a Metal rasterizer (rasterizer.zig) that composites the shaped runs and the rest of the page's geometry onto the backing store. Behind those two boundaries the engine's own code never touches font internals or the GPU API surface beyond a narrow command layer.

Three decisions keep paint cheap enough to run every frame. The first is that shaping is memoized. Profiling put Core Text typesetting at the top of layout, not paint, because line breaking measures every candidate substring it tries; the memo answers those measurements from a hash table instead. An entry is keyed by the text together

with everything that can change its metrics—family, size, weight, slant, letter and word spacing—and returns width, ascent, and descent. That key is the correctness argument: each field is there because two runs differing in it must not share an entry, and a bold run is measured in its real face rather than synthesized. Entries own copies of their text and family, because the caller's slices point into a layout arena reset underneath the cache—the arena divergence of Section 2, answered here by copying—and the table is bounded, so a page generating endless distinct strings cannot grow it without limit.

font_registry.zig repeats the move. The platform layer memoizes one Core Text font per @font-face file, keyed by a hash of its bytes; taken at the lookup, that hash is paid once per measurement, so the line breaker above re-hashes every webfont for every candidate substring—77% of one site's main-thread load time. Hashing once, at registration, makes the lookup constant-time: put the cost where the input changes, not where it is read.

The second decision is that the surface holds a band of the document rather than the window. The rasterizer paints a strip the height of the viewport plus a fixed overscan above and below it, and records the strip's top edge in document space as band_top. A scroll that stays inside the strip needs no new pixels: it is a change of blit origin, and the shell samples the same bitmap at a different offset. Only a scroll carrying the window out of the band asks for a frame.

The band is recomputed rather than scrolled. Nothing shifts the existing pixels: when the band moves, the engine translates the fragment tree to the new band's origin and repaints the surface there. That is more work than a blit of the retained region plus a strip, and it is chosen deliberately, because not everything on the page moves with the content. Sticky and fixed boxes re-pin against the new viewport offset, and a fixed-attachment background anchors to the viewport rather than the document, so shifting retained pixels would carry exactly those elements along with the content that should have scrolled past them. Repainting at a translated origin has one rule where a shift would need two. The same reasoning governs invalidation: band_top describes the pixels in one bitmap that is not swapped when the page is, so every path that changes what is on screen marks the band invalid.

Frame skipping is the third decision. Paint is driven by the shell's display loop, which asks for a frame on every run-loop turn, but cw_view_render returns whether it actually produced one, and the engine produces a frame only when the backing store has changed. log_timing reports frames=13 and skipped=756 on a typical load: 13 frames emitted, 756 elided as unchanged. Frame skipping is not an optimization layered on a correct renderer; together with the band it is what lets the engine idle at full page complexity without paying a paint cost for frames nobody sees.

IV. Layout

Layout is organized as one mode per CSS formatting context, and the modes are small enough that each is a tractable module: block, inline, flex, grid, and table, with multicol and sticky positioning handled by smaller passes on top. None is a reimplementation of a browser's layout engine in miniature; each is the specific algorithm for its context—block flow, line boxes, flex distribution, grid track sizing, table layout—written against the shared fragment tree.

What makes the decomposition hold is that the modes agree on an interface rather than an implementation. A mode is handed a node, a containing block, and the constraints the parent context imposes; it produces fragments in the layout arena and reports a used size. It does not know which mode laid out its parent or which will lay out its children, so a flex item containing a table is two modes meeting at one geometry contract rather than a special case in either. Because the modes share that contract and one arena discipline (Section 2), adding a formatting context is adding a module, not extending an engine, and the cost of getting one wrong is bounded by the boxes it owns.

Layout knows about the viewport in exactly one place. content-visibility: auto lets a box skip laying out its contents when it is far enough off screen, which requires layout to ask where the painted band (Section 3) is. The engine answers with the box's retained size—the geometry the previous full-document layout published—and skips only boxes whose used width does not depend on their contents; a flex or grid item's width is content-dependent, so skipping it would change the size it reports to its parent and move the page. Skipped contents generate no fragments, so paint and hit-testing never see a box layout declined to compute. One property and one guarded predicate aside, layout is a pure function of the document and the containing block—which is what makes the determinism reported in Section 7.1 achievable at all.

The striking fact about the layout code, though, is how small it is relative to the rest of the style system. css_values.zig, the module that parses and represents CSS values, is larger than any single layout mode—larger, in fact, than block, inline, and table put together. That is not an accident of this engine; it is where the bulk of a CSS engine goes. Layout is a few dozen well-understood algorithms, each bounded by the geometry of its box model: the number of ways to distribute free space along a flex line does not grow when the language gains a property. The cascade, by contrast, is a combinatorial surface. Every property has a syntax to parse, valid values to represent, a computed form to resolve against font and viewport units, an inheritance rule, an interpolation behaviour, and a shorthand that may expand into it—and the code admitting all of those combinations is what swells. Nothing in it is hard the way grid track sizing is hard; it is hard

because it does not compress, each property being its own small specification and there being hundreds of them. That reframes the cost of a CSS engine for anyone considering the same project: the intimidating part is not laying out a page but deciding, for every property on every element, what value it should have—and that part is not deep, it is wide, which is a different kind of expensive and a different kind of tractable.

V. Instant Back: the Parked Tree

The history cache is the paper's central technical contribution, and its design is the opposite of the usual one. A conventional back-forward cache stores the document—the DOM, the style sheets, the JavaScript state—and, on restore, re-runs layout to rebuild the geometry it needs to paint. That re-layout is what makes restore cost anything at all: the cache saved the document, not the work of turning it into a picture. Turtle parks the laid-out fragment tree instead. Restore does not re-layout; it reuses the geometry that layout already produced, so the expensive pass is skipped entirely. The fragment tree is the natural unit for this because it is already the thing paint walks (Section 2): parking it means the engine can go straight back to painting a page whose geometry was never discarded.

The interesting half of the design is the safety argument, and it is one sentence: parking frees nothing. A conventional cache must decide, when it evicts or when a background worker fails, whether the page it holds is still safe to show. That decision is a dilemma whenever the failure is ambiguous—a paint worker that died mid-frame leaves a cache holding a page it cannot guarantee is coherent. Turtle's cache sidesteps the dilemma by construction. Parking only moves the fragment tree into the cache; it does not free or mutate any of it. The page is exactly as valid after parking as it was before. A failed paint-worker join therefore stops being a question about the cache's contents at all; it is a question only about the worker. Only eviction frees, and eviction is a deliberate, synchronous act that happens when the ring is full or memory pressure drops it. Because nothing is freed by parking, the cache never has to reason about whether a parked page is still good—it always is, until eviction takes it away.

A. What "the page" is, and why the walk is symmetric

Parking a page means quieting everything that page can run, and a page is not one frame: it is the root frame, that frame's dedicated workers, its descendant frames recursively—each iframe has its own JavaScript context and therefore its own task queue—and the same enumeration again for each popup it opened. A timer anywhere in that tree keeps a "parked" page allocating as effectively as one in the top document, so a walk that stops at the root frame parks nothing at all on a page whose script lives in an iframe.

The engine has exactly one definition of that set, walkFrames, and freeze and thaw are the same function over it, distinguished only by a compile-time boolean that selects suspend or resume. This is not stylistic. The walk has legs—descendant frames, and then popups—and while those legs were written out at each call site, deleting one from thaw alone compiled cleanly, leaving a restored page's popups suspended forever with no diagnostic anywhere. Under a single visitor parameterized by direction, an asymmetric freeze/thaw is not expressible: the same code reaches the same schedulers on the way down and on the way back up, because there is only one traversal.

Two exclusions from the walk are deliberate. Shared workers are shared across pages by definition, so suspending one to park an opener would stall a live page still connected to it; a shared worker outliving a parked page is correct, not a leak. Already-closed frames are excluded because their schedulers were reset on teardown and nothing can queue into them again.

Having one definition paid for itself immediately. The pagehide and pageshow events are per-window, but the engine had grown a second, narrower notion of "this page" that fired them at the top window only—so an iframe got a pageshow on its own load and then nothing on a park or a restore, both a specification deviation and exactly the drift a single definition is meant to prevent. Folding that dispatch onto walkFrames closed it.

Firing script from inside the walk leaves one residual hazard. A pagehide handler may synchronously close a popup, mutating the very list the loop is iterating, so a popup can be visited twice or skipped. It is not memory-unsafe—frames are arena-allocated, so the stale slice stays readable—and the root frame is safe by construction, because it is walked to completion before the popup list is read at all. But a duplicated or skipped pagehide on a popup is a real residual defect.

B. Exclusions, triggers, and one refusal

Not every page is parked, and the exclusions are narrow. A page mid-load is not parked, because its fragment tree is still changing and parking it would freeze a snapshot of a moving document. A page holding a live socket is not parked, because the connection outlives the page and parking would strand it. A page served with no-store is not parked, because the response header is an explicit instruction that the content must not be cached. These three are the whole exclusion list; every other page is a candidate.

A parked page is not frozen. Three conditions force a restyle on restore, and each is a reason the parked geometry might no longer match what the page should show. A viewport change means the page must be laid out for a different size, so the parked geometry is rebuilt. A backing-scale change—the window moving between screens with different densities—changes the rasterization

even when the geometry is identical, so the page is re-painted at the new scale. And the DOM can drift while the page is parked; the engine detects this with a RenderSignature, a digest of the document's structure, and a mismatch between the signature recorded at park time and the one at restore time triggers a restyle. These triggers are the entire restyle surface; absent all three, restore reuses the parked tree as-is.

The design has one refusal. The engine asks stillRestorable before every restore, and the answer can be no. When the answer is no, the engine falls back to a normal load rather than pretending the parked tree is usable. The refusal is what keeps the fast path from ever serving stale geometry: it is not that a parked page is guaranteed to be reused, but that when it is reused, it is reused correctly.

Why that check happens at restore rather than at park is the part of the design most easily gotten wrong, and it follows from a fact recorded in the code: freezing a page is not a script barrier. Suspending a scheduler gates queued work—timers, microtasks, anything that goes through a queue. It does not gate a listener invoked directly off a network completion, and at least one path does exactly that: an XMLHttpRequest's load listener is dispatched straight into the JavaScript engine when the transfer finishes, with no scheduler anywhere in the call. The eligibility check does not close this either, because an in-flight request is not an open socket—the exclusion counts long-lived sockets and event streams, not transfers. A page can therefore be parked with a request outstanding and run script when it lands.

What a parked page can still do follows directly: mutate its own DOM, making the parked tree a picture of a document that no longer exists, or navigate itself, committing a replacement page for the same frame and leaving the entry's borrowed URL pointing into a dead arena. Neither event arrives with any notification on the cache's side of it, which is why a flag set by a handler cannot work—there is no handler to set it.

So stillRestorable is a witness comparison made at restore time. At park the entry records the addresses of the page and the document its trees were built from, as bare integers that are never dereferenced—the whole question they answer is whether dereferencing would be safe. At restore it compares them, rejects an entry whose navigation has committed but not yet swapped, and re-runs the full eligibility check, so a socket opened or a no-store response received while parked refuses the restore exactly as it would have refused the park. The result latches: an entry that stops being restorable never becomes restorable again, so a Back button the shell has already been told is unavailable cannot flicker back.

One thing is deliberately not a refusal. A merely mutated DOM does not throw the page away, because it is recoverable: the signature mismatch turns the restore into a restyle rather than a blit. Refusal is reserved for the cases where the parked tree describes a page that is gone, not for the cases where it describes a page that has changed.

The fast path is not free, and the measurement shows where it stops paying. On small and medium documents parking makes restore effectively instant and the speedup over the restyle fallback is large (Table II); on the largest document it stops paying altogether. We do not claim a bounded restore cost. The design claim is that restore skips layout; the assumption that skipping layout is what makes restore fast does not hold above a document size Section 7.2 reports and analyses. Parking is a win on the pages a history cache is for—the ones users actually go back to—and a documented ceiling on the ones it is not.

VI. Conformance

The engine is measured against the standard browser test suites before it is measured on real sites, because conformance is the floor the evaluation sections assume. Turtle scores 100 on Acid 3[4], the long-standing rendering and DOM conformance suite. On html5test[5] it scores 381 at the start of the work and 425 at the end—a gain attributed commit-by-commit in the repository rather than claimed as a single jump.

Neither number should be read for more than it measures. Acid 3 is a fixed, finite page: passing it establishes that a broad slice of DOM, CSS, and script behaviour works together on one document, which is a real floor and not a ceiling. html5test counts feature presence against a checklist; a point means the engine answers the probe, not that the feature is correct under load, and a checklist score is exactly the metric that rewards the shallow implementation Section 8 is built to catch. Both are reported because they are comparable across engines and cheaply reproducible, not because they settle the question. Web Platform Tests[6] would be the stronger instrument, and we run parts of it, but cannot report a total: the css-grid suite aborts on the panic in Section 9, so grid conformance is unmeasured rather than measured and poor.

The engine has no WebRTC, no service workers, no MathML, and no support for most media codecs. Each is a subsystem with its own surface—a peer-to-peer transport, a background-execution model, a math layout mode, a codec pipeline—that would roughly double the code under measurement without advancing the paper's question, which is whether a single author can reach a usable engine. Each is also separable: none is on the path from a document to pixels, which is what the originality claim in Section 1 is scoped to.

One gap is a different kind of absence. Content Security Policy level 1 is not unimplemented in Turtle; it is unimplementable as written. CSP 1 requires gating eval at a choke point, and the V8 binding this engine links against exposes no AllowCodeGenerationFromStrings hook. Without it there is no single place to refuse code generation. The distinction matters because a partial implementation

would be worse than none: the directives that can be enforced would report a policy in force while the one that matters most is unenforceable, and a security mechanism right about most of its surface is not one. This is a property of the binding, not the engine, which is why CSP 1 appears in the gap list rather than as a missing feature—and why the fix is an upstream API rather than a subsystem to write.

VII. Evaluation

A. Apparatus and per-suite method

All numbers in this section come from the committed harness (scripts/bench.sh, engine 60e0c69d3) on an M4 Pro. Pages are served from local snapshots over loopback, so what is measured is the engine and not the network: a benchmark that includes a live fetch measures the weather. Every site is run 5 times after a discarded warm-up, and each suite reports median and p95 from those runs, together with the minimum and maximum. The median is what the prose quotes; the p95 is what tells us whether the median means anything, and both are read from the same CSV columns the tables are generated from. No figure in this section is transcribed by hand: the tables and every number in the surrounding text are emitted from the raw CSVs by a generator, and a checker fails the build if the two disagree or if a number appears in prose without one.

A site that does not render produces a row of NA and stays in the table rather than being dropped, because a site that fails to render is a result, not an outlier; the harness also records how many retries it spent before giving up, so a failure that is intermittent can be told from one that is total.

B. Load and layout

Table I lays out 22 sites, from 9 ms on example (a 338-px document) to 1779 ms on github-zig. Two rows carry the section. First, rfc9110: 144,586 px laid out in 1467 ms—a document an order of magnitude taller than anything else in the table, laid out in a time that is merely large, not pathological. Second, github-zig is the slowest site at 1779 ms despite a content height of only 1,588 px. Cost tracks script and DOM complexity, not document size: a small page with heavy scripting costs more than a huge page that is mostly static text.

The tails support that reading. The tall-but-static document is also the stable one: rfc9110's p95 of 1481 ms sits close beside its median, which is what a workload dominated by a deterministic layout pass looks like. The small, script-heavy page is the volatile one: github-zig's p95 of 2089 ms is well clear of its median, which is what a workload dominated by script execution and its scheduling looks like. The same contrast that shows up between the two medians shows up again in their spreads, from an independent column.

Layout is deterministic. content_h is byte-identical across all runs of every site, so the geometry layout produces does not vary run to run; the variance in load time is scheduling, not layout. That licenses using a single content height per site elsewhere in the paper, and it separates the two sources of variance cleanly: whatever moves the median between runs, it is not the layout. mdn-flex failed all 5 runs—after 12 retries—and appears as NA. It stays there rather than being dropped.

C. History-cache restore

Table II reports the fast path and the restyle fallback separately, because a blended mean would describe neither. On small and medium documents the fast path is effectively instant and the speedup is large. In the last row it collapses to parity with the restyle fallback, the paper's most important negative result.

On rfc9110 the parked restore takes 2055 ms against the restyle fallback's 2059 ms: a gap of 4 ms between a path that skips layout and a path that does not. Figure 1 shows the two converging.

The first question about a parity result is whether it is an artifact of picking the median, and here it is not. Across the 5 runs the two distributions overlap outright: the fast path's slowest run (2060 ms) is slower than the restyle path's fastest (2052 ms), and the restyle path's p95 of 2074 ms is within the same band. Any statistic drawn from these runs reports parity, because the runs themselves are not separable.

The second question is what the parity means, and the answer comes from comparing scaling rather than levels. Between wikipedia-zig and rfc9110 the document grows 10.1×. Over that step the restyle fallback—which does the full work, restyle and layout and paint— goes from 239 ms to 2059 ms, a factor of 8.6×: close to linear in the document, which is what a layout-dominated path should do. The parked fast path over the same step goes from 12 ms to 2055 ms, a factor of 171×. The path that skips the expensive pass scales worse than the path that performs it. That is the finding, and it is stronger than "the speedup goes away on big pages": two curves with those slopes must cross, so the ceiling is structural rather than a constant that could be tuned away.

A third reading rules out the comfortable explanation. One might suppose the gap is small because layout on this document is cheap, but the load table refutes that directly: laying out rfc9110 takes 1467 ms, which is far more than the 4 ms that separates the two restore paths. The fallback is doing at least that much work more than the fast path and finishing at effectively the same time. So the two are not trading comparable amounts of layout—they are both dominated by some cost they share, which swamps the difference. The design claim survives this (restore does skip layout, and the load table prices what it skips); the performance assumption does not.

The paper reports the ceiling and names the suspects rather than guessing at a cause: full-document paint,

geometry republish, and the two walks in republishParkedGeometry, all of which scale with the document and none of which parking removes. A profile would settle which; we have not run one, and we do not claim to know. The result stands either way, and it bounds the contribution: parking makes restore instant on the documents a history cache is for, and above roughly 14k px it buys nothing.

Two caveats printed by the harness are carried verbatim wherever restore latency is reported. First, cw_view_last_restore_ms is an integer counter, so a median reported as zero means at or below its resolution, not zero—which is why the fast column's speedups on the smallest documents are written as lower bounds rather than ratios. Second, the restyle rows' content_h is measured at zoom 1.25—the zoom is what forces the restyle—so rfc9110's restyle height of 160,318 px is not comparable with its fast-path height of 144,687 px, and no ratio in this paper is taken across the two.

D. Parked-page memory

Table III reports RSS bracketed by drop_cache inside one process, so the delta is the parked pages rather than the difference between two differently warmed processes.

The result the ring depends on is that per-page cost is flat as the ring fills. Parking one ziglang costs 9.7 MB; with the ring full at 3, the per-page figure is 9.2 MB. Parking one wikipedia-zig costs 121.9 MB; at depth 3 the total is 358.2 MB, or 119.4 MB per page. Nothing amplifies as pages accumulate: the second and third parked pages cost what the first did. That is the property the cap rests on, because it makes the cache's worst case a multiplication the engine can reason about—ring depth times the heaviest page—rather than an unknown that has to be discovered by filling the ring. 121.9 MB for a single heavy article is precisely why that product is capped at 3 and dropped under memory pressure.

Two qualifications. At the light end the measurement is at its floor: example parks for 0.7 MB, and RSS sampling cannot resolve a per-page cost that small, so the apparent decline in its per-page column should be read as "indistinguishable from zero", not as a saving. And the delta is not the peak: filling the ring with wikipedia-zig transits 1,106 MB of resident memory before settling. A cap justified on steady-state cost still has to be paid for at the transient.

E. Fidelity of the corpus

Serving from snapshots removes the network from the measurement, but it introduces a question the network did not have: whether the snapshot is still the page. The harness answers it explicitly rather than by assumption. A fidelity suite loads each site twice in the same engine—once from the live URL and once from the local snapshot—and compares the content heights, recording the signed difference per site.

TABLE I
Load and layout, 22 sites. mdn-flex failed all 5 runs and appears as NA.

Site	Median (ms)	Content height (px)
example	9	338
cern-first	12	552
danluu	168	6,829
sqlite	18	1,958
curl	37	1,935
kernel	28	1,157
apache	21	2,574
postgres	420	6,093
openbsd	25	1,230
ziglang	55	4,162
rust-lang	59	3,367
python-org	146	4,353
nodejs	115	1,091
hackernews	76	1,654
lwn	48	6,496
arxiv	196	6,977
mdn-flex	NA	NA
rfc9110	1467	144,586
whatwg-intro	628	27,390
wikipedia-zig	307	14,394
wikipedia-main	197	11,919
github-zig	1779	1,588

Of 22 sites, 9 reproduce the live page's content height exactly, to the last fractional pixel. 3 still drift by more than 130%, the worst by 203.1%. Those rows are named individually rather than averaged in: a snapshot that renders half again as tall as the page it came from is measuring something, but not the site it is named after. A further 2 sites are recorded as uncompared rather than as matching—one because the live fetch redirected to a different host than the one requested, so the harness refused to compare two pages it could not prove were the same page, and one because the snapshot crashed the engine locally while the live page loaded. Both are reported as gaps in the corpus.

F. Methodology of the measurement itself

These are the errors that would have produced plausible wrong numbers, and each was caught by the fidelity check above rather than by inspection. Snapshots served without their stylesheets rendered 2,560 px against 3,935 px live—a difference small enough to look like a rendering discrepancy rather than a missing resource. Sites with no snapshot silently measured the 404 page, with four sites reporting an identical 186.88 px, which is the shape of that bug: the giveaway was not that a number looked wrong but that several sites agreed too closely. Rate-limited image mirroring produced zero-height images. And the apparatus changed its subject: an un-throttled fetcher got the host blocked by one origin, and another returns 403 to non-browser clients, so the corpus was quietly becoming a set of error pages. Each of these is a way to publish a confident number for a page that was never rendered, and none of them would have been visible in the timing data alone.

TABLE II
History-cache restore: fast path vs restyle fallback, reported separately because a blended mean would describe neither.

Site	Fast (ms)	Restyle (ms)	Speedup
example	0	3	≥3×
ziglang	0	30	≥30×
sqlite	1	26	26×
python-org	3	85	28×
wikipedia-zig	12	239	20×
rfc9110	2055	2059	1×

TABLE III
Parked-page memory: RSS bracketed by drop_cache inside one process, so the delta is the parked pages.

Site	1 parked (MB)	3 parked (MB)	Per page (MB)
example	0.7	0.7	0.7 → 0.2
ziglang	9.7	27.7	9.7 → 9.2
wikipedia-zig	121.9	358.2	121.9 → 119.4

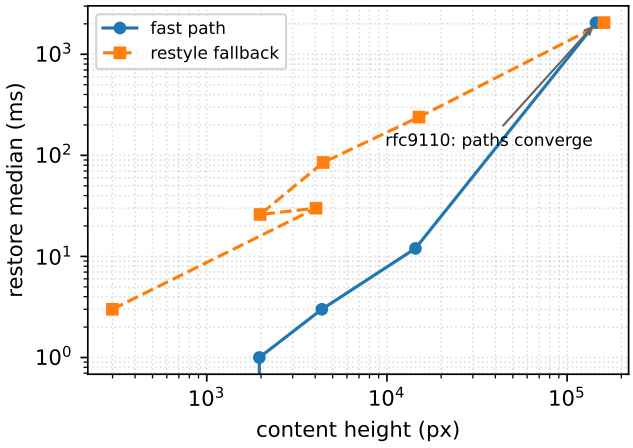

Fig. 1. Restore cost vs document size: fast and restyle median against content height, log-log. On the largest document (rfc9110) the two paths converge.

VIII. Methodology

The engine's architecture and its proof of concept were written by hand. The agent loop was adopted later, at a specific inflection point: with the design settled, the work that remained was conformance—the WPT[6] CSS layout suites, tens of thousands of tests, each one a small, fully specified change with a mechanical pass or fail. That is the shape of work a loop is good at and hand-writing scales badly on. The architectural decisions this paper claims as contributions sit on the hand-written side of that boundary; the volume sits on the other.

The loop is run by cwcode, a coding agent written in Go and authored by the paper's author, structured as a pipeline: user requirement → dev spec → implementation → verification & testing → code review. Every item in every stage carries an identifier, and a traceability matrix links each requirement to the spec that refined it, the implementation that satisfied it, and the verification that checked it. The artifacts of the loop—the specs, the recorded findings, and the committed harness—are inspectable, so the claim is checkable rather than asserted.

The identifiers are what make the loop more than a sequence of prompts. A requirement no spec refines is an unmatched row; a spec item with an implementation but no verification is a half-filled one. Neither is something a model notices about its own work, because each is a property of the whole pipeline rather than of the file in front of it. Making the linkage explicit lets a mechanical pass ask the question the model cannot: not "does this code look right" but "did anything that was asked for go unchecked".

The loop is not tied to a single model. Over the project's life it ran on Claude 4.6, then Claude 4.8, then DeepSeek V4 Pro, and then DeepSeek V4 Flash 0731; no one of them is responsible for the work as a whole. What carries the method across those substitutions is the pipeline, the identifiers, and the verification stage, none of which is model-specific. Context length is the one model property the method does lean on: a window large enough to hold a layout mode, its tests, and the spec item that motivated it lets the model edit code it can still see rather than code it is recalling.

cwcode's own contribution to the loop is hash-anchored editing. read_file annotates each line with a three-hex content hash, and edit_lines takes ranges with the expected hashes and rejects the whole batch on any mismatch. The edit path therefore never depends on character-perfect reproduction of a line—an error the model would otherwise make—because the tool refuses to apply a change whose target has drifted. A search-and-replace edit tool instead asks the model to reproduce existing text exactly—a requirement no model meets reliably, and one that fails silently when the reproduction is subtly wrong: the edit lands somewhere plausible, or does not land, and nothing says so. Anchoring on a hash moves the check from the model to the tool, converting a silent misapplication into a loud refusal the loop can retry, and it makes batched edits safe: a batch is accepted only if every anchor still holds, so an edit computed against a stale read cannot half-apply. The lineage of the technique is Can Akay's "hashlines"[7].

The evidence for the loop is what it caught. Over the course of the project the verification stage surfaced, among others:

- thirteen vacuous assertions—tests that passed whether or not the code worked;
- a CSP hole: deleting policy inheritance left the whole suite green while an iframe could escape its embedder's policy;
- a leak checker that exited 0, so "the leak checker will catch it" was never actually a test;
- an aliasing bug where _coords at offset zero made

p.coords === p;

- a silent-zero test filter that printed "N of N passed" while running nothing;
- a window missing .fullScreenPrimary, which made toggleFullScreen: a silent no-op, found only by driving the real app.

The list has a shape. Four of the six are not defects in the engine at all; they are defects in the evidence that the engine worked. A vacuous assertion, a leak checker exiting zero, a filter that runs nothing, and a suite that stays green after a security property is deleted are one failure wearing four costumes: a green result that was never contingent on the code being correct. An agent loop is unusually good at producing that failure, because a model rewarded for a passing suite converges on a passing suite, and the cheapest way to pass is to test less. So the verification stage cannot be "run the tests"—it has to attack them, deleting the behaviour under test and requiring the suite to notice. The other two are the complementary case: real defects no suite reached, found only by driving the shipped application, which is why the loop keeps a stage that exercises the real app rather than a harness.

Two limits should be stated. This is one project by one author, with no control arm: we cannot say what the conformance work would have cost written by hand throughout, and we do not claim the loop is faster. And the defect list is the set the loop caught, which says nothing about the set it missed—Section 9 is the visible part of that remainder. What the evidence supports is narrower: a loop whose verification stage is adversarial toward its own tests finds a class of defect a build-and-test pipeline structurally cannot.

IX. Limitations and Future Work

- Restore is not instant on very large documents. The parked fast path collapses to parity with the restyle fallback above roughly 14k px (Section 7.2); a profile of the two walks in republishParkedGeometry is the natural next step.
- A reproducible out-of-bounds panic in subgrid layout (grid.zig:2331, empty col_sizes) aborts the WPT[6] css-grid suite. It predates the measured build; the release binary loads the page cleanly, and the panic is reached only through the harness reftest path.
- 55 leaking tests, all in CSS parsing.
- mdn-flex fails to render and appears as NA in the load table.
- macOS/Apple Silicon only. AppKit, Core Text, and Metal are not portable.
- Ad-hoc signed, so Gatekeeper blocks the binary without a Developer ID.
- No WebRTC, service workers, MathML, or most media codecs, and CSP 1 is unimplementable against this V8 binding (Section 6).
- file:// navigation is unsupported.

Two of these cost the paper more than the others. The subgrid panic does not merely fail a suite—it aborts one, so the engine's css-grid conformance is unmeasured rather than measured and poor; no claim here rests on a grid figure, and none should be read into its absence. The leaking parser tests are the second: they run and pass, so they are evidence about behaviour but not about lifetime—and this engine's documented crashes are lifetime bugs (Section 2). A leak checker excluded on the tests most likely to exercise the arena boundary is exactly the gap Section 8 warns about, found in our own work.

Portability, likewise, is a boundary rather than a to-do: shaping, rasterization, and the shell are written against Core Text, Metal, and AppKit, so a port is a re-implementation of Section 3 and not a build-system change.

Future work follows directly from the ceiling in Section 7.2: profiling the parked-restore path to find where skipping layout stops paying, and fixing the subgrid panic so the WPT css-grid suite runs to completion. Both are measurements the harness can already take once the code permits them.

References

[1] Z. S. Foundation, "Zig programming language," 2016. [Online]. Available: https://ziglang.org/

[2] T. C. Project, "Browser architecture," 2008, the Chromium Projects design overview. [Online]. Available: https://www.chromium.org/developers/design-documents/

[3] A. Grosskurth and M. W. Godfrey, "A case study in web browser architecture," in Proceedings of the 2005 International Workshop on Software Architecture and Metrics (SAM), 2005, a structural study of the Mozilla, Internet Explorer, and Konqueror code bases.

[4] W. S. Project, "Acid3 browser test," 2008, the Acid3 rendering and DOM conformance suite. [Online]. Available: https://www.webstandards.org/action/acid3/

[5] N. Leenheer, "html5test," 2010, hTML5 feature conformance test suite. [Online]. Available: https://html5test.com/

[6] W3C and contributors, "Web platform tests," 2013, cross-browser test suite for the web platform. [Online]. Available: https://web-platform-tests.org/

[7] C. Akay, "hashlines," 2026, blog.can.ac, 2026-02-12. [Online]. Available: https://blog.can.ac/2026/02/12/the-harness-problem/