Beyond WebAssembly: Rethinking the Web Browser with Post-JavaScript, VM-Free Page Execution

Brent Hartshorn

July 8, 2026

Abstract

The web's active-content model rests on JavaScript and, beneath it, on WebAssembly—a stack virtual machine that, in current browsers, must cross a JavaScript boundary for every access to the document. We report a web browser built from the ground up on the minipy/ShivyCX substrate, in which a page's active content is either Python, executed by an embedded interpreter against a live document, or *native machine code* compiled ahead of time from an `rpython` `<script>` and invoked through `ctypes`—with no virtual machine, no JavaScript wrapper layer, and no third-party browser engine. Concretely: `<script type="python">` runs against a shared `document`/`console`/`window`; `<script type="rpython">` is compiled (`py2c` → `gcc -O2`) to a cached per-page shared object callable by `ctypes`, while the *same source* still runs on stock CPython; a `<canvas>` is drawn by a JIT-compiled native fragment shader with time and pointer uniforms at one native call per frame; and JavaScript runs by *translation onto the same interpreter and document*, not a second VM. Nothing from Blink, WebKit, Gecko, Servo, or NetSurf is used; pages render through a small software widget layer into a Wayland framebuffer. We position the result as a systems alternative to Redox OS (which ports NetSurf or Servo) and ChromeOS (which *is* Chrome/V8/Blink), and as a descendant of TempleOS's ground-up philosophy— arguing that each of these optimizes *within* the JavaScript/engine boundary, whereas co-designing the page-execution model lets us move it.

1 Introduction

Three prior reports in this line of work push a single thesis—that high-performance, self-contained execution comes from co-designing every layer at once—down through the stack: a foundational critique of the compiler/OS boundary [1], a restricted-Python dialect compiled whole-program to native C and machine code [2], and a toolchain (front end, transpiler, assembler) that compiles and runs on the runtime it produces [3]. The one layer those reports leave untouched is the one users actually see: the browser, and the execution model of the pages it runs.

That model is JavaScript, and beneath it WebAssembly [4]. WebAssembly is a real advance—a compact, sandboxed, near-native bytecode—but it is a *stack virtual machine* placed alongside the JavaScript engine rather than in place of it: it has its own linear memory, its own warmup, and, decisively, no direct access to the document. In current browsers every DOM read or write from WebAssembly marshals through a JavaScript wrapper. The web thus carries two execution engines where the page author wanted one, and native code—the very thing WebAssembly exists to deliver—reaches the page only at arm's length.

Our substrate is written in Python, and the choice is deliberate. We take the quality of a programming language to be measured by how completely it lets one rebuild one's own tools with it—up to and including the language itself. By that measure Rust, scores poorly for this program:

reinventing `rustc` is not a single-author undertaking, and the language surface is large enough that re-deriving it from memory is itself hard work. (Newer systems languages such as Zig and C3 share this property: the toolchain is not casually reinventable by one person.) Python inverts the trade. It is slow by default, but its *reinventability* is total: one can write a C compiler in it, re-express the interpreter in a restricted subset and compile that subset to native code, add a JIT, an assembler, and—here—a browser and an operating-system-shaped runtime, each in the same language and each able to rebuild the last. The standard objection, that Python is slow, is answered not by a benchmark but by the existence of this stack: slowness is an engineering problem the substrate already solves [2], while reinventability is the property no faster language in our experience preserves.

This paper applies that discipline to the browser. Rather than optimize *within* the JavaScript / engine boundary, we co-design the page-execution model itself. Active content is Python on an embedded interpreter, sharing one live document; native page code is an ordinary shared object compiled ahead of time and called through `ctypes`, not a sandboxed VM; and JavaScript, where present, is *translated* onto the same interpreter and document rather than hosted in a second engine. The renderer beneath all of this is a small software widget layer targeting a Wayland framebuffer—no third-party engine, no GPU stack.

We report four results on a working browser built this way. (1) **Python as the page language**: `<script type="python">` drives a live document with dynamic element creation, first-class event handlers, and two-way input binding. (2) **Native page code without a VM**: an `rpython <script>` is compiled ahead of time to a cached native `.so` and called via `ctypes`—a faster, leaner, and CPython-compatible alternative to WebAssembly, verified end to end inside the browser. (3) **Native drawing**: a `<canvas>` filled by a JIT-compiled fragment shader with time and pointer uniforms, one native call per frame. (4) **JavaScript on the same engine**: a source translator places plain `<script>` onto the same interpreter and document. We then compare against Redox OS, ChromeOS, and TempleOS, and close with an honest account of what is demonstrated versus sketched.

2 The substrate, in brief

We recap only what makes the browser's results legible; the substrate is developed in full elsewhere [2, 3].

Programs are written in a deterministic, type-inferable subset of Python— "rpython" in the `py2c` sense—and translated by the `py2c` source-to-C transpiler into idiomatic C under a three-tier object model: unboxed C scalars where whole-program inference proves monomorphism, monomorphic structs with a static type descriptor, and a tagged-union box only where genuine polymorphism remains. A back end (ShivyCX) then adds passes that cross the language boundary—register-file partitioning for cheap context switches, call-graph *contracts* that license fallback-free SIMD, and bare-metal data placement—because the whole program's types, call graph, and register needs are known at once. The same dialect also has a bytecode interpreter, minipy, for running rpython dynamically; `py2c` compiles that interpreter to native code like any other program.

Paper 3 [3] closed most of the toolchain into this one language: the parser front end compiles to native through the same pipeline; `py2c` itself compiles and *begins* to run on minipy; and an in-dialect assembler encodes ShivyCX's output byte-identically to GNU `as`. Only the linker remains external. The consequence for what follows is that the substrate is not a fixed host but a *movable, self-rebuilding* one—a property we now spend at the top of the stack rather than the bottom.

Two of its facilities are load-bearing here. First, the browser *embeds* minipy: the interpreter that runs a page's Python is the substrate's own interpreter, compiled to native and linked into the

browser binary by `py2c`. Second, native page code reuses `py2c` directly—an `rpython <script>` is compiled by the *same* transpiler that compiles the runtime, so page code and runtime are citizens of one pipeline rather than two. In the prototype the emitted C is finished by a stock system compiler; removing that last compiler is precisely the self-hosting trajectory of Paper 3, not a new problem.

3 Native page code without a virtual machine

The page ships native code in a `<script type="rpython">` block. On load the browser compiles it—`py2c` to C, then a C compiler to a position-independent shared object, cached per page—and the page's Python (running on the embedded interpreter, §4) loads it and calls into it through `ctypes`. The contrast with WebAssembly is structural. Where wasm delivers near-native code inside a second stack machine that reaches the document only through JavaScript glue, here the delivered artifact is an ordinary native `.so` in the browser's own process, entered by an ordinary C-ABI call. There is no bytecode, no virtual machine, and no wrapper layer; native code is a first-class citizen of the page rather than a sandboxed guest. Because the object is produced by `gcc -O2`, it uses the machine's real registers and SIMD directly—the thing a stack interpreter structurally cannot do.

The design also keeps a property WebAssembly cannot. The *same page source runs unmodified on stock CPython.* The author writes an ordinary `ctypes.CDLL(...)` and an ordinary call such as `dll.calc_sum(1, 2)`, so under a normal Python interpreter, with the `.so` on disk, the call resolves through real `ctypes` and returns the native result; the browser merely *redirects* the path to a per-page cache. A page that uses native code is therefore portable to any host that speaks Python and C—which is not something one can say of a wasm module bound to browser imports.

Compilation is the cost, and it is paid once. Each block is compiled on first visit and cached under a per-page directory keyed by a hash of its source; an unchanged block on reload is a cache hit with no recompilation, and per-page directories keep caches from colliding across sites. The time and memory spike of running the transpiler and C compiler is thus confined to that first compile, rather than amortized across the page's lifetime as a resident JIT and its warmed code would be. This inverts the wasm/JIT memory profile, in which the engine and its generated code stay resident for as long as the page is open.

The demanding half is that the embedded interpreter must *itself* perform the `ctypes` call when it runs a page's Python natively, and two obstacles stood in the way. First, `py2c`'s ctypes support is a *static* bridge: it resolves foreign symbols at link time, not by runtime `dlopen`. The interpreter therefore gained genuine runtime FFI—a small C shim providing `dlopen`, `dlsym`, and an indirect call through the resolved function pointer, reached through that same static bridge, guarded so the interpreter still imports cleanly under CPython, and linked into every build of the interpreter. Second, minipy has no `__getattr__`, so an expression like `dll.calc_sum(1, 2)` cannot dispatch on the library handle at run time. We resolve this at compile time: the pass that lowers a page's Python to minipy bytecode rewrites `ctypes.CDLL(path)` and `dll.name(args)` onto the FFI shim and redirects the path to the page's cache. The rewrite touches only the interpreter path—the unrewritten source is exactly what runs under CPython.

We verify the whole path end to end and headless: the browser loads a page whose `<script type="rpython">` defines a trivial `calc_sum`, compiles and caches its shared object, boots the page's Python on the embedded interpreter, and on an event calls the native function and reports its result— the value returned is the one the native code computed. The mechanism is deliberately general: because the artifact is a real shared object and the call is a real C call, the same path is what lets native code *draw* to the page (§5). We do not claim a complete replacement for a browser's compiled-code tier. We claim something narrower and, we think, more useful: native

page code need not be a virtual machine at all. An ordinary object, compiled ahead of time and called by ordinary means, is fewer layers to reach, cheaper to keep resident, and portable to any host that speaks C.

4 Python as the page language

A page's active content is Python, carried in `<script type="python">` and executed by the embedded interpreter against a live document—the familiar `document`, `console`, and `window`. That document is a single object owned by the interpreter and shared by every kind of page code: the same one native code reaches (§3), that the canvas draws into (§5), and onto which JavaScript is compiled (§6). This is already a departure from the mainstream browser, in which each scripting path is a separate binding onto a C++ DOM; here there is one document, in one language, for the whole page.

The load-bearing design decision is that the document is *authoritative inside the interpreter*. Page code mutates ordinary in-language objects— creating elements, appending children, assigning event handlers, setting values—and the renderer never writes them back. After each event the browser asks the interpreter to serialize the current document to a compact JSON form, reparses that, and rebuilds the on-screen widget tree from scratch. The renderer is therefore a pure function of the serialized document and holds no DOM state of its own. This keeps the browser→interpreter interface to a single read-back of a string, and it is what lets Python, translated JavaScript, and native code all address one document without a per-language binding layer: whoever mutates the interpreter's objects, the next render simply reflects them.

The document is genuinely live and bidirectional. Elements are created and appended at run time and appear on the next render; event handlers are first-class values, so a handler the page assigns to an element is invoked when that element is activated, with events reaching back through opaque handles so the renderer never holds a reference into interpreter memory; and text typed into an input flows back into the document, so a script that reads a field sees what the user actually typed. Each of these—dynamic construction, handler dispatch, and two-way input—is verified headless by driving the real event path and reading the resulting document back.

None of this required a second DOM implementation. Because the document is one interpreted structure re-rendered on demand, extending the page with a new kind of content is a matter of giving that content access to the structure, not of building a new binding: native code writes results a script stores back, the canvas of §5 is simply one such element, and the JavaScript of §6 is lowered onto the same objects. The document is the shared substrate of the page, as the dialect is the shared substrate of the toolchain.

5 Native drawing: a canvas fragment shader

A `<canvas>` is drawn by native code. Its shader is an `rpython <script>` compiled by exactly the path of §3—the page's own native shared object—exposing a function `render(buf, w, h, t, mx, my)` that fills a `w`×`h` buffer of packed pixels. The signature is a fragment shader's: alongside geometry it receives a *time* uniform `t` and a *pointer* uniform `(mx, my)`, so one function expresses a static image, an animation, and a cursor-reactive effect. It is the seed of a Canvas2D/WebGL surface reached without a graphics stack: no GPU driver, no separate shading language, and no JavaScript drawing API—only native code writing a framebuffer, in the same dialect as the rest of the page's native code, produced and called by the mechanism already in hand.

The interface is designed so that a frame is a single native call. The browser allocates one pixel buffer in native memory, reused across frames, and each frame invokes `render` once to fill it; the pixels never enter the interpreter. The on-screen canvas widget then blits that buffer into the window's framebuffer through a pointer view over the same memory—a bounded copy, not per-pixel foreign calls. An earlier design called the shader once per pixel and was correct but wasteful; folding the loop into the shader and passing the buffer by reference reduced a frame from thousands of boundary crossings to one. The pixel path thus stays entirely in native code, from the shader's writes to the blit, with the interpreter uninvolved.

Animation required giving the browser a clock. Its event loop was, like a document viewer's, purely reactive: it woke only on input. We added an opt-in frame loop. While an animated surface is on screen, the windowing layer requests a frame callback from the compositor, and on each callback advances `t`, delivers the current pointer position, refills the canvas, and repaints; when no such surface is present the loop stays off and the browser is again purely event-driven. The opt-in is deliberate—a page with no animation should not spin—and it leaves the rest of the system unchanged when nothing is animating.

We verify the shader off the compositor in two ways: a headless check confirms that a single `render` call fills the whole buffer and that its output varies with both the time and the pointer uniform, and the identical shader, run under CPython through real `ctypes`, renders a frame sequence to an image—the same native output the browser blits, made viewable without a display server. Live animation itself is observable under a Wayland compositor; headless, it is established structurally and by that off-target rendering. The result closes the loop opened in §3: the native-code path is not only for computing values a script reads back but for producing the page's pixels directly, at one native call per frame, with neither a virtual machine nor a graphics stack in between.

6 JavaScript on the same engine

JavaScript runs, but not in an engine of its own. A plain `<script>` is parsed to a syntax tree and *translated* at load time into the same restricted Python the interpreter already runs, against the same `document`. We reuse the parser on which the Js2Py project builds [7, 8] and walk its tree, emitting minipy for the common shape of page scripts: functions, `var`/`let`/`const`, the usual control flow, member and call expressions, the operators (with `===` mapping to `==` and `&&` to `and`), object literals as dictionaries, and the familiar array methods. The output is ordinary page Python; a translated handler and a hand-written one are indistinguishable to the document. JavaScript thus becomes one more source language feeding the single interpreter, not a second runtime beside it.

This is deliberately a subset, not a JavaScript virtual machine. The goal is parity for *document scripting*—the DOM calls, handlers, and arithmetic ordinary pages use—not the whole of ECMAScript. Constructs outside the covered subset are left untranslated rather than mistranslated, so an unsupported script simply does not run; and the harder corners of the language—implicit coercion in `+`, hoisting, closures, prototypes, `this`—are out of scope by design. What the subset buys is that the two page languages share one engine and one document, so nothing in §3–§5 needs a JavaScript-specific path: a translated script creates elements, binds handlers, and reads inputs through the very objects Python does.

The direction is worth naming, because the obvious comparison runs the other way. Pyodide, and frameworks such as PyScript built on it, bring Python *to* the web by compiling CPython to WebAssembly and running it inside the existing browser [9]; there, Python is a guest of the JavaScript/wasm stack, reaching the document through JavaScript bindings and atop a multi-megabyte VM image. We invert the arrangement: the browser *is* native Python, and JavaScript

is the guest—lowered onto the native interpreter at load time and sharing its document, with no virtual machine added for either language. One arrangement carries an entire interpreter into the incumbent stack; the other makes the incumbent language a source dialect of a native one. It is the move of the toolchain papers applied to the page: reduce the languages to one engine rather than stack a new engine on the old.

7 Comparison and related work

The dominant answer to "native speed on the web" is to compile to the browser's existing engine. Emscripten first compiled C/C++ to a JavaScript subset (asm.js) that engines could specialize [10], and WebAssembly [4] replaced that subset with a proper bytecode. Both are advances, and both keep the shape we question: the compiled code runs inside the JavaScript engine—in Chrome's case V8, which executes JavaScript and WebAssembly alike and hands neither direct access to the document, only bindings generated into the engine [11]. Native code thus arrives on the page as a *guest of the JavaScript runtime*. Our design removes the host: compiled code is a native shared object in the browser's own process, and the document is an object it can be handed, not a foreign surface reached across a binding layer.

ChromeOS takes the same premise to the scale of an operating system. Its principal user interface *is* the Chrome browser [12]: a Linux kernel underneath, the Ash/Aura window manager around it, and Blink and V8 at the center running the web platform. The Chromebook is therefore the JavaScript-engine web delivered as a computer—an impressive result, but one that inherits the engine's every assumption, including JavaScript as the language of interaction and a JIT as the mechanism of speed. It does not rethink the page-execution model; it ships it. Where ChromeOS makes the browser the computer, we make the substrate the browser: the same whole-program-compiled Python runtime that Papers 1–3 push to the metal is what runs the page.

Redox OS is the closest in spirit on the systems side, and instructive in the difference. It rebuilds the operating system from scratch in Rust—microkernel, drivers, C library, display server—a genuinely ground-up effort [5]. Yet at the browser it reaches for existing engines: its shipping browser is NetSurf, which has its own layout engine but no full JavaScript [13, 14], and the path to a modern web is a port of Mozilla's Servo, which as of late 2025 runs only in rudimentary form—rendering simple pages and crashing on the second [15]. The lesson we draw is not that this work is misdirected but that rewriting the OS does not, by itself, rethink the browser: the engine is a second mountain, and porting one preserves its execution model wholesale. We attack that model directly and accept a smaller browser as the price.

The nearest ancestor in philosophy is TempleOS, Terry Davis's from-scratch operating system with its own compiler, graphics, and language in one coherent, dependency-minimal whole [6]. Paper 1 already took it as a model of what a single author holding an entire stack can build. We share its conviction that a system becomes comprehensible and reinventable when it refuses external dependencies and speaks one language throughout, and we differ in target and trade-off: where TempleOS is an offline personal computer whose language is paired with an unoptimizing compiler, we aim the same ground-up, single-language discipline at the *web* layer, with a language compiled whole-program to native code and, per Papers 2–3, on a path to compiling itself.

A single thread runs through the comparison. asm.js, WebAssembly, ChromeOS, and the Redox browser ports all optimize *within* the boundary the incumbent web draws—JavaScript and its engine on one side, everything else reached across it. Pyodide (§6) moves Python to the same side of that boundary without moving the boundary. Our contribution is to relocate it: one interpreter and one document, native code as a first-class citizen rather than a sandboxed guest, and JavaScript

admitted as a source dialect rather than a resident runtime. The result is smaller and less complete than any engine named here; it is also dependency-free by construction, and it shows that the boundary was a choice, not a necessity.

8 Limitations

The model's directness has a cost we state plainly: native page code runs in the browser's own process, called through `ctypes`, with no sandbox. The very property that lets it reach the document without a binding layer also removes the isolation WebAssembly is built to provide. As it stands the design suits trusted or first-party code and controlled deployments—the setting the foundational critique of Paper 1 already assumed—rather than arbitrary untrusted pages; a trust boundary, whether compile-time verification of the `rpython` or an out-of-process sandbox, is future work and does not follow for free from anything here. Relatedly, the native path still finishes through a stock C compiler: `py2c` emits C, and `gcc` produces the shared object. Removing that last compiler is not a new problem but exactly the self-hosting trajectory of Paper 3, whose one remaining external tool is the linker.

The rest of the ledger is scoped engineering. The JavaScript translator is a subset aimed at document scripting, not conformant ECMAScript. The renderer is software, with no GPU path; the canvas is filled on the CPU. The foreign-call interface is presently narrow—integer arguments and a pixel buffer—so richer signatures await widening. Navigation is over local pages; a URL fetcher exists but is not yet wired into it. Live animation is observable under a Wayland compositor and, headless, is established structurally and by off-target rendering. None of these is a research obstacle, and none changes the paper's claim, which is about where the execution boundary sits, not about feature completeness.

9 Conclusion

The whole-stack thesis of this line of work has, until now, pointed downward—from a restricted-Python dialect to native C, to the metal, and through the toolchain that produces it. This paper spends that substrate at the top of the stack. A working browser runs a page's Python on an embedded, whole-program-compiled interpreter against one live document; compiles a page's `rpython` to a native shared object and calls it through `ctypes`, with no virtual machine and with the same source still running on stock CPython; draws a canvas with a JIT-compiled native shader at one call per frame; and admits JavaScript by translating it onto the same interpreter and document. Nothing from a third-party browser engine is used, and nothing but a stock C compiler stands between the page and self-hosted native code.

The point is not that a small browser can be written from scratch—many can—but *where* the boundary ends up when it is. WebAssembly, ChromeOS, and the Redox browser ports each accept JavaScript and its engine as the fixed center and optimize around it; we moved the center, and found a page-execution model that is native, VM-free, and dependency-free by construction. That we could is, finally, an argument for the language. A good language is one in which you can rebuild your own tools, up to and including itself: Python lets us write a C compiler, re-express and compile its own interpreter, add a JIT and an assembler, and now a browser and its execution model—each in one language, each able to rebuild the last. Its default slowness was an engineering problem the substrate solved; its reinventability is what made the rest possible.

Source code: https://github.com/brentharts/ShivyC/tree/master/examples/rpython2c/minibrowser

References

[1] B. Hartshorn (2025) Foundational Problems with Compilers and Operating Systems. https://ai.vixra.org/abs/2507.0081

[2] B. Hartshorn (2026) Rethinking Meta-Interpreters for High-Performance Execution. https://ai.vixra.org/abs/2606.0084

[3] B. Hartshorn (2026) Closing the Compiler Loop: Toward a Self-Hosting, Dependency-Free Python JIT. https://doi.org/10.5281/zenodo.21178004

[4] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, JF Bastien (2017) Bringing the Web up to Speed with WebAssembly. PLDI '17, pp. 185–200. https://doi.org/10.1145/3062341.3062363

[5] Redox OS. *a complete Unix-like microkernel-based operating system written in Rust* https://www.redox-os.org/

[6] D. M. Božović (2024) *TempleOS: Architecture and Principles of Lightweight Operating System Development.* https://doi.org/10.13140/RG.2.2.34698.27845

[7] P. Dabkowski. pyjsparser: a fast JavaScript parser in pure Python. https://github.com/PiotrDabkowski/pyjsparser

[8] P. Dabkowski. Js2Py: a JavaScript-to-Python translator. (with ES6-ES8 support by G. Cosman 2026) https://github.com/OpenSourceJesus/Js2Py

[9] Pyodide: the Python scientific stack, compiled to WebAssembly. https://pyodide.org/

[10] A. Zakai (2011) *Emscripten: an LLVM-to-JavaScript Compiler. OOPSLA '11 Companion;* https://doi.org/10.1145/2048147.2048224

[11] Google (Chrome for Developers) *What is Blink?* (Blink uses V8 to execute JavaScript and WebAssembly; the DOM is exposed through generated V8 bindings.) https://developer.chrome.com/docs/web-platform/blink

[12] *ChromeOS.* Wikipedia (uses the Google Chrome browser as its principal user interface; Ash/Aura window manager on a Linux kernel). https://en.wikipedia.org/wiki/ChromeOS

[13] Redox OS. *This Month in Redox — April 2025* (NetSurf is the current browser and lacks full JavaScript; SpiderMonkey and Servo porting underway). https://www.redox-os.org/news/this-month-250430/

[14] NetSurf: a free, open-source web browser with its own layout engine. https://www.netsurf-browser.org/

[15] *Servo ported to Redox* (October 2025). OSnews. https://www.osnews.com/story/143714/servo-ported-to-redox/