

Neural Inference of Realtime Audio in the Web: Porting Anira to WebAssembly

Anton von Raumer
Audio Communication Group
Technische Universität Berlin
Berlin, Germany
anton@vonraumer.dev

Fares Schulz
Audio Communication Group
Technische Universität Berlin
Berlin, Germany
fares.schulz@tu-berlin.de

Abstract—While recent advances in browser technologies such as WebAssembly, the WebAudio API with its AudioWorklets, and SharedArrayBuffers have enabled low-latency, multi-threaded audio processing in the browser, real-time neural inference remains challenging due to their strict timing constraints and inter-thread communication overhead. This work presents a port of the Anira C++ neural inference library to the web, extending its build process for WebAssembly compilation, and implementing a TypeScript wrapper package with a custom thread management strategy, as well as support for custom pre- and post-processors and inference backends, and integration with AudioWorkletProcessors. We analyze the performance of the resulting Anira Web npm package with a benchmarking suite across Chrome, Firefox, Safari, and native execution, analyzing the influence of execution environment, model architecture, custom inference backends and pre/post-processor, using linear mixed-effects models. We achieve stable real-time performance across all three major browsers, with Safari and Firefox approaching native runtimes more closely than Chrome, and user-defined JavaScript components incurring a browser-dependent cost that never breaks real-time bounds, compared to their WebAssembly-compiled counterparts. We demonstrate the feasibility of real-time neural inference in the browser and provide a practical building block for deploying audio machine learning models on the web.

Index Terms—webaudio, webassembly, neural inference, real-time audio, onnx runtime, audioworklet, c++, javascript, audio processing

I. INTRODUCTION

Audio processing possibilities on the web have increased drastically over the past one and a half decades, paving the way for complex audio applications directly in the browser. Until 2009, the only way of computing on the web was simple JavaScript code on the main thread, competing with all other tasks running there, such as rendering the user interface. This was solved when Web Workers became broadly supported, making it possible to run computation on separate threads [1]. However, communication between threads was limited to message passing, whose overhead and latency made it unsuitable for real-time audio processing. Furthermore, audio APIs were browser-specific, like the Audio Data API in Firefox [2] or the WebKit Audio API in Safari and Chrome. This was remedied by the Web Audio API, proposed in 2011 [3], a standardized way of creating audio graphs and processing audio in the browser. It achieved wide adoption by the mid-2010s, and

was finalized in 2021 [4], [5], but custom processing nodes once again ran on the main thread. The AudioWorklet API, introduced in 2018 and widely adopted by 2021 [6], then allowed developers to run custom audio processing code on the audio thread directly, and the SharedArrayBuffer allowed data to be shared between threads directly, without message passing [7].

One more major development made more complex audio processing feasible on the web: WebAssembly (Wasm), introduced around 2017 [8]. Wasm is a low-level binary format that serves as compilation target for languages like C/C++ and Rust, and can be executed in the browser. Its benefits include small file sizes and fast load times, sandboxed security, portability, and near-native performance. Compilers like Emscripten [9] augmented WebAssembly with glue code for interacting with JavaScript, spawning Web Worker threads, or accessing other web APIs.

This combination of technologies enabled low-latency audio processing on the web in a standardized way across all major browsers for the first time, and developers soon ported or developed audio libraries and applications, from domain specific languages like FAUST [10] to full-fledged digital audio workstations [11].

At the same time, neural networks proliferated across many fields, audio processing among them, from audio classification to audio synthesis. Offline processing of audio files in batches is trivial, but many applications require real-time processing of audio streams, preferably with low latency, and merely piping samples through a network in sequence will quickly lead to high latency and dropouts, as the network cannot keep up.

The Anira C++ library was developed by Schulz and Ackva [12] to solve this problem, using a novel approach: It runs multiple instances of the inference engine in parallel in a thread pool, each processing a discrete chunk of samples. Anira automatically batches the incoming audio samples to fit the input shape of the network, and ensures that the output is retrieved in the correct order, even if different batches of inference finish at different times. Anira supports three inference engines: LiteRT (formerly known as TensorFlow Lite) [13], ONNX Runtime [14], and PyTorch [15]. While it can be used on Windows, macOS, and Linux, it is not yet supported on the web.

In this work, we therefore port Anira to the web browser. The result consists of two layers: first, an extension of Anira’s build process to support WebAssembly compilation via Emscripten, making it available as a dependency for C++ projects targeting WebAssembly; and second, a TypeScript wrapper package that exposes Anira’s functionality directly to JavaScript, ready to be integrated as an `AudioWorkletProcessor` in the WebAudio API. Since WebAssembly exports only flat functions with numeric arguments, C++ classes cannot be exported directly; the first layer therefore also includes a C API over Anira’s C++ classes, through which the second layer reaches the module.

A key challenge is the choice of inference engine to bundle within the WebAssembly module; we evaluate which of Anira’s three supported engines is best suited for this purpose and provide a static WebAssembly build of the selected engine as a reusable artifact [16]. A central design decision of the TypeScript wrappers is a custom threading strategy, since Emscripten’s built-in pthread support would prevent key features of the wrapper, such as user-provided pre- and post-processing functions and custom inference backends written in JavaScript (Section III-C). The wrapper is published as an npm package¹.

To evaluate the result, we run a benchmark suite built on the measurement principles of the existing Anira benchmarks across native and WebAssembly targets, the latter in all major browsers. Using linear mixed-effects models, we analyze the influence of key factors on runtime performance, including execution environment, model architecture, and whether pre- and post-processors and inference backends are compiled into the WebAssembly module or provided from JavaScript.

All source code, including the benchmark suite and analysis scripts, is available in the supplementary repository [17]; the C++ changes to Anira have been incorporated into the upstream repository with version 2.1.0. A live demo is available at <https://anira-project.github.io/anira-web-example/>.

II. RELATED WORK

Compiling an existing C/C++ audio library to WebAssembly behind a JavaScript/TypeScript API is an established pattern: `Essentia.js` [18] brings the `Essentia` music-analysis library to the browser, and the `FAUST` ecosystem compiles its signal-processing language to WebAudio nodes [10], [19]. These target feature extraction and signal-processing primitives, and where `Essentia.js` performs inference it delegates to `TensorFlow.js`. Anira instead treats neural inference as its central concern, built around real-time safety, decoupling inference from the audio callback with explicit latency management so that inference cost need not cause audio dropouts.

General-purpose in-browser inference has matured through libraries such as `TensorFlow.js` [20] and, more recently, engines like `WebLLM` [21]. Studies of in-browser inference analyze and reduce its gap to native execution, attributing the gap to underused SIMD and WebAssembly runtime overhead

[22], [23]. We measure this overhead specifically for hard-real-time audio, where per-block deadlines, rather than aggregate throughput, are the binding constraint.

Native real-time neural inference is provided by Anira [12], which this work extends, and by `RTNeural` [24]; the convolutional and recurrent models we benchmark are representative of this domain, which spans amplifier and effect emulation [25], [26] and neural synthesis [27]. To our knowledge, no prior work provides a general-purpose library for real-time neural audio inference in the browser; Anira Web fills this gap with a custom threading model.

III. IMPLEMENTATION

A. Choice of Inference Engine

On the web, an inference engine can be made available to Anira in two ways: compiled into the WebAssembly module alongside Anira, or supplied at runtime from JavaScript through Anira’s custom-backend mechanism (Section III-D).

We bundle an engine into the WebAssembly module so that C++ projects targeting WebAssembly get a statically linked inference backend out of the box, without a separate engine or any JavaScript; so that JavaScript users avoid crossing the JavaScript↔WebAssembly boundary on every inference call; and so that performance stays directly comparable to Anira’s native execution.

For the bundled engine to be suitable, it must be available under a permissive open-source license, support common neural-network operations, and, most importantly, be usable in C++ codebases that compile to WebAssembly, meaning a static library build targeting Wasm must be available or buildable. Other inference engines for the web can, to the best of our knowledge, only be used from JavaScript, even if they do run using WebAssembly; this is the case for `LiteRT` as well, whose guides on use in a C++ codebase compiling to WebAssembly are outdated and no longer supported [28]. The only engine matching all criteria was `ONNX Runtime`: actively maintained by Microsoft, available under the MIT license, with well-documented instructions for building a static library targeting WebAssembly.

The second option, supplying the engine from JavaScript, builds on the wrapper package’s mechanism for providing inference backends from JavaScript (Section III-D), letting developers plug in a different or updated engine without rebuilding the WebAssembly module. We demonstrate it with a JavaScript-side `ONNX Runtime` backend using the `onnxruntime-web` package, pinned to the same `ONNX Runtime` version (1.19.2) and configured with the same session options as the bundled build, so that the two differ only in whether inference is dispatched inside the WebAssembly module or from JavaScript.

B. Building ONNX Runtime for WebAssembly

On native platforms, Anira links a prebuilt `ONNX Runtime` shared library; for the WebAssembly target, however, the engine must instead be available as a static library, which the linker pulls into the single WebAssembly module alongside

¹<https://www.npmjs.com/package/@anira-project/anira>

Anira. ONNX Runtime does not ship prebuilt WebAssembly static libraries, and those builds must target the very same Emscripten version as the final module, so the static library had to be built from source, a lengthy process that needs to be repeated for each new version of the ONNX Runtime and of Emscripten. Therefore, we automated it using Github Actions [29], and host the resulting builds on Github as release artifacts.

When Emscripten is detected as the compiler, Anira’s build process now selects this WebAssembly-targeted ONNX Runtime static library, the compiler flags `-matomics`, `-mbulk-memory`, and `-msimd128` are added (atomics and bulk memory for shared-memory threading, 128-bit SIMD), and link flags are set for Emscripten; the complete flag sets are pinned in the supplementary repository [17]. Within the Anira codebase, the predefined `__EMSCRIPTEN__` macro guards source-level adjustments.

The browser downloads the resulting `AniraWeb.wasm`, 13.5 MB (2.0 MB Brotli-compressed) in the `wasm-release` build and dominated by the statically linked ONNX Runtime, plus 0.23 MB of JavaScript glue and about 0.1 MB of wrapper; models are supplied by the application. The JavaScript-side ONNX Runtime backend additionally needs the 11.0 MB (1.9 MB Brotli) `onnxruntime-web` module, which the benchmarked revision inlines into the wrapper bundle. Two further link flags make the module import the shared `WebAssembly.Memory` of Section III-C (`IMPORTED_MEMORY`, `SHARED_MEMORY`), created with 512 MiB (537 MB) as both initial and maximum size by the wrapper package, and with 1 GiB (1074 MB) by the benchmarks of Section III-E: the heap is never reallocated, so the typed-array views each context holds onto it never need refreshing while audio is being processed. The figure is a ceiling; memory consumption is not measured in this work (Section VI).

C. Thread Management

Anira’s threading model runs a pool of inference threads in parallel, into which audio blocks are fed through its process function, typically from a dedicated real-time audio thread. On the web, the real-time audio thread maps directly to the WebAudio rendering thread inside an `AudioWorkletProcessor`, and the inference threads to Web Workers.

Emscripten can map pthreads to Web Workers automatically, but this approach has three limitations that make it unsuitable here. First, it requires the worker pool size to be fixed at compile time, whereas the desired number of inference workers may vary at runtime. Second, its Wasm Audio Worklets API [30] requires the audio processing callback to be defined in C++ and compiled into the module, with no way to customize it from JavaScript at runtime. Third, JavaScript-defined callbacks registered with Emscripten can only be invoked from the main thread, so custom pre- and post-processors or JavaScript-provided inference backends could

not be called from within the inference workers where they are needed.

The foundation of our approach is a single shared `WebAssembly.Memory` instance, backed by a `SharedArrayBuffer`, created on the main thread and passed explicitly to each context that requires it: inference workers and the `AudioWorklet` alike. Because every context loads the Wasm module over this same memory, they all share one heap, so a C++ object created in one context is reachable from any other by its pointer, without serialization or copying. Each context still needs its own stack, which we allocate within the shared memory from the main thread before starting it. Objects are constructed on the main thread and used elsewhere by pointer, so the workers hold no long-lived state of their own and memory ownership remains unambiguous.

The inference workers and the `AudioWorklet` are all brought up the same way: each loads its own instance of the Wasm module over the shared memory, adopts its stack region, reaches the C++ objects it needs by their pointers and installs a JavaScript callback hook through which C++ can dispatch into custom JavaScript functions running in it (Section III-D).

Inference workers are plain Web Workers, spawned and torn down on demand. Each one drives the shared `InferenceThread` object, running Anira’s inference loop: it polls the job queue with an exponential backoff strategy until told to stop, and signals job completion through either a semaphore or an atomic flag, depending on the `blocking_ratio` parameter. Any custom JavaScript inference backend runs inside this worker. Because all contexts share one heap, the job queue and completion flags are the very same C++ structures Anira relies on natively, compiling to WebAssembly atomic operations on the shared memory, so Anira’s existing thread-safety guarantees carry over to the browser unchanged.

The `AudioWorklet` runs on the WebAudio rendering thread. It drives the `InferenceHandler`, calling its process function on each incoming block to submit it and collect completed results, and it is where custom JavaScript pre- and post-processors run. One web-specific caveat concerns the `blocking_ratio` parameter, which lets the audio thread optionally block during the process callback, waiting for results up to a configurable fraction of the buffer duration. The deadline for this wait is computed with `std::chrono::steady_clock`, which Emscripten maps to `performance.now()`; since that API is unavailable in the `AudioWorklet` global scope, a `Date.now()` polyfill is installed automatically, degrading timer resolution to one millisecond.

D. TypeScript Wrapper Package

A wrapper package exposes the runtime described so far to JavaScript and TypeScript without writing any C++. The module’s C API, compiled only for the Emscripten target, provides one `extern "C"` function per C++ class method (e.g. `inferencehandler_process`), with objects passed

as pointers into the shared heap. Emscripten’s Embind could generate such a layer, but it binds JavaScript objects to one module instance, whereas here objects are created in one context and reached by raw pointer from others (Section III-C), which plain exported functions taking pointers serve directly. The package mirrors Anira’s C++ classes with corresponding TypeScript classes; each wrapper holds a numeric pointer to its underlying C++ object and forwards its method calls to the exported C functions for that object.

Instantiating a wrapper can either allocate a new C++ object and store its pointer, or attach the wrapper to an already-existing pointer without allocating anything; the latter is how objects created on the main thread are picked up again inside the inference workers and the AudioWorklet.

The central entry point is the `AniraWeb` class, which owns the module instance and the shared memory that every wrapper needs in order to call into the C++ objects. It exposes the wrapper constructors as factory methods that inject both into every object they create, mirroring the original C++ API. `AniraWeb` also drives the threading described above: spawning and tearing down inference workers, setting up the AudioWorklet, registering JavaScript backends and pre- and post-processors, and the memory sharing and stack hand-off.

Custom JavaScript functions are registered in the context that will call them, so they run directly on the right thread with no cross-thread message passing: developers write their own inference-worker and AudioWorklet processor classes and register JavaScript inference backends in the worker, and pre- and post-processors in the worklet. When C++ needs to run a custom function, it fires the context’s callback hook with the pointer of the relevant object, and the receiving context looks that pointer up in its local registry and dispatches to the matching implementation.

E. Benchmarks

To evaluate performance across browsers and against native execution, the benchmark logic is implemented in a single C++ source file that compiles to both a native binary and a WebAssembly module. For the WebAssembly target, model files are embedded in the virtual filesystem at compile time using Emscripten’s file embedding [31], keeping the benchmark self-contained. Both targets are built at `-O3` without `-ffast-math`, natively with Apple Clang 17.0.0 and for WebAssembly with Emscripten 4.0.23 and the flags of Section III-B, statically linking ONNX Runtime 1.19.2 built with SIMD and threading enabled. The single resulting binary runs unchanged in all three browsers, and since SIMD instructions are part of WebAssembly’s validation rules [32], successful instantiation confirms SIMD availability. Runtime assertions and C++ exception catching remain enabled, so browser-versus-native differences are, if anything, slightly overestimated.

Each run times the processing of successive audio buffers, filled with random samples, for one combination of model, buffer size, inference backend, and pre- and post-processor (Section IV-B). Inference is dispatched to a single inference

thread, so each recorded per-block time reflects the round-trip latency of one inference in isolation; in real usage, Anira would run multiple inference threads in parallel, a configuration these benchmarks do not exercise (Section VI). Inference backends and pre- and post-processors written in JavaScript can be provided in place of their C++/WebAssembly counterparts, using the extension mechanisms of Section III-D, to measure the cost of lifting components outside of the compiled WebAssembly module.

In the browser, the timed loop runs in a dedicated Web Worker standing in for the AudioWorklet, with inference in a separate inference worker as in practice; timing inside an `AudioWorkletProcessor` is not feasible with only the millisecond `Date.now()` polyfill of Section III-C.

IV. METHODS

To evaluate Anira Web, we formulate three research questions. **RQ1** concerns platform overhead: how much does the per-block processing time increase in the browser compared to native execution, and does performance remain within real-time bounds across the three major browsers? **RQ2** concerns cold-start behavior: the original Anira evaluation observed a first-iteration spike for certain configurations [12]; is the same startup penalty present in the browser, and does it differ across browsers and native execution? **RQ3** concerns the cost of JavaScript-provided components: what is the performance penalty of supplying custom pre- and post-processors and custom inference backends from JavaScript, rather than bundling them into the WebAssembly module?

A. Models

The benchmarks were run using two different models, both of which are provided by Schulz and Ackva in the Anira example repository [33]. The first is `SteerableNAFX`, a Temporal Convolutional Network (TCN), consisting of a number of TCN blocks, each containing a causal convolution layer, a PReLU activation layer, and a residual connection. The second is `GuitarLSTM`, which uses a hybrid architecture, consisting of two 1-D convolutional layers, followed by a stateless LSTM (long short-term memory) layer, followed by a dense layer. The two 1-D convolutional layers act as feature extracting down-sampling, while the dense layer maps the final LSTM output to a single audio sample.

Both models require custom pre- and post-processing: `SteerableNAFX` for maintaining a receptive field context of 13332 samples; and `GuitarLSTM` for constructing the 150-sample context window expected by the model, batched across inference iterations as a sliding-window. Computationally, both consist only of buffer management and copying between Anira’s ring buffers and the model’s tensors, without resampling, filtering, or spectral transforms; this cost is measured, together with Anira’s framework overhead, by the bypass runs of Section V-A.

These two standard Anira examples represent two common architecture families in real-time audio processing, a pure convolutional and a hybrid recurrent network with different

compute profiles; both are lightweight models designed for low-latency real-time use.

B. Benchmark Configurations

All benchmarks were run on a MacBook Pro with an Apple Silicon M3 Pro chip, 18 GB of RAM, running macOS Tahoe 26.6.2.

Each benchmark run is defined by a combination of five factors: *environment*, *model*, *buffer size*, *inference backend*, and *pre/post-processor*. Four execution *environments* were tested: native execution, serving as the reference, and the three major browsers Chrome 153.0.8010.52, Firefox 156.0, and Safari 26.6.2. Each *model* is benchmarked at three *buffer sizes*, 128, 1024, and 8192 samples. The *inference backend* factor has four levels, two backend types each implemented once in C++ (compiled with the benchmark to native code or WebAssembly, and the only *backends* available natively) and once in JavaScript (outside the WebAssembly module, browser only). The bypass *backends* (BypassBackend, JSBypassBackend) copy input tensors directly to output without performing any inference, so that bypass runs isolate the non-inference pipeline cost (Anira’s framework together with the model’s pre- and post-processing); the ONNX *backends* run inference with the ONNX Runtime bundled with Anira (ONNXRuntime) or through the onnxruntime-web package (ONNXRuntimeWeb), configured to match the bundled *backend* (Section III-A) so that the two are directly comparable. The *pre/post-processor* of each *model* is likewise provided in two implementations: a C++ version (SteerableNafxPrePostProcessor and HybridNNPrePostProcessor) and a complete JavaScript reimplement of the same logic (JSSteerableNafxPrePostProcessor and JSHybridNNPrePostProcessor), available only in the browser.

Each run comprises ten *repetitions* of fifty *iterations*. Within a *repetition*, the inference handler is constructed once and then fed fifty consecutive audio buffers, each processed and timed individually, simulating the block-by-block processing of a live audio stream. A *repetition*, by contrast, is an independent cold-start run: after its fifty *iterations* complete, the inference handler, pre/post-processor, and configuration are destroyed and rebuilt for the next *repetition*, capturing initialization overhead and run-to-run variability. Anira’s optional warm-up inferences are left at the library default of none, so *iteration* 0 of every *repetition* contains the session’s first inference, as in the original evaluation [12]; module instantiation, worker start-up, and session creation, by contrast, precede the timed loop. Each run therefore yields 500 timed measurements.

Browsers deliberately coarsen their high-resolution timer as a Spectre mitigation [34]. Table I shows the `steady_clock` timer resolution reported for each platform by a single probe at benchmark start, an upper bound on the effective resolution. For most configurations this is negligible, but for the shortest buffer tested (128 samples, 2.90 ms at 44.1 kHz) the coarsest resolution (20 μ s) represents roughly 0.7 % of the real-time

TABLE I
STEADY_CLOCK TIMER RESOLUTION PER PLATFORM, AS REPORTED BY A SINGLE PROBE AT BENCHMARK START; AN UPPER BOUND ON THE EFFECTIVE RESOLUTION.

Platform	Timer resolution
Native	84 ns
Chrome	10 μ s
Firefox	20 μ s
Safari	20 μ s

threshold. Each measurement spans, for a single block, the interval from handing it to the process function until its result is available, timed on a Web Worker rather than the rendering thread (Section III-E); since inference runs in a Web Worker in either case, the dominant cost is measured under production conditions.

Two datasets were compiled from the measurements. The *RQ12 dataset* isolates platform overhead and therefore contains only the C++ *backends* and *pre/post-processors*, which run in all four *environments*, of the form $4 \times 2 \times 2 \times 3 \times 10 \times 50$ (*environment* $\times$ *backend* $\times$ *model* $\times$ *buffer size* $\times$ *repetition* $\times$ *iteration*), for $N = 24,000$ measurements. The *RQ3 dataset* addresses the cost of JavaScript-provided components in the three browser *environments*, covering all four inference *backends* and both *pre/post-processor* implementations, of the form $3 \times 4 \times 2 \times 2 \times 3 \times 10 \times 50$ (*environment* $\times$ *backend* $\times$ *pre/post-processor* $\times$ *model* $\times$ *buffer size* $\times$ *repetition* $\times$ *iteration*), for $N = 72,000$ measurements. In both datasets, runtime is normalized as runtime per sample (*RpS*, in μ s/sample), the measured processing time of one buffer (one *iteration*) divided by the *buffer size*, for comparability across *buffer sizes*.

The exact Anira revision used is pinned as a Git submodule in the supplementary repository [17].

C. Statistical Analysis

Descriptive statistics for the bypass-only runs (Anira Web’s non-inference overhead) are computed using the `pastecs` package [35]. Tail behavior per configuration (standard deviation, coefficient of variation, 99th percentile, maximum, and deadline-miss rate) is likewise reported descriptively.

Two linear mixed-effects models (LMMs) are fitted using the `lme4` package [36] in R [37], with restricted maximum likelihood (REML) estimation and ANOVA using Kenward-Roger degree-of-freedom correction via `lmerTest` [38]. One LMM is fitted to each dataset: *LMM12* to the *RQ12 dataset* (RQ1 and RQ2), *LMM3* to the *RQ3 dataset* (RQ3). For both LMMs, log-transformed *RpS* is the dependent variable, and a random intercept per *repetition* accounts for between-repetition variability. *Iteration number* is treated as a categorical fixed effect with sum contrasts, enabling post-hoc tests of individual *iterations* against the grand mean rather than a linear trend.

In *LMM12*, the fixed effects are *backend type*, *buffer size*, *iteration number*, *environment*, and *model*, entered with their full five-way interaction. In *LMM3*, the fixed effects are *inference backend*, *pre/post-processor*, *model*, *buffer size*, browser

environment, and iteration number, with all main effects and interactions up to five-way (62 model terms).

Post-hoc comparisons of estimated marginal means, averaged over all remaining factors, were computed using the `emmeans` package [39] with Holm correction for multiple comparisons. *Environment* comparisons (RQ1) and *backend* and *pre/post-processor* comparisons (RQ3) use pairwise contrasts; *iteration* effects (RQ2) are examined by contrasting each *iteration* against the grand mean, following the approach of Schulz and Ackva [12]. Statistical significance is assessed at $\alpha = 0.001$ throughout; reported p-values are nonetheless bracketed at the 0.05, 0.01, and 0.001 levels.

V. RESULTS

A. Library Performance

We evaluate the bypass *backend* (Section IV-B) to confirm that Anira Web’s non-inference overhead is compatible with real-time processing independently of inference cost. The real-time threshold per sample ($RTT = 1/f_s$) defines the available budget; at 44.1 kHz, $RTT \approx 22.68 \mu\text{s/sample}$. Across all tested *environments* and both *models* (Table II), mean bypass *RpS* remains well below the *RTT*, with the largest mean of $1.03 \mu\text{s/sample}$ (Chrome, GuitarLSTM) representing less than 5% of the budget. This figure is the price of the context-window copying described in Section IV-A together with the framework’s own overhead, and confirms that Anira Web’s non-inference overhead does not constrain real-time feasibility. Table II reports arithmetic means over all *buffer sizes* and is therefore dominated by the 128-sample runs, where native bypass is in fact marginally slower than Chrome; the estimated marginal means of Fig. 1, being geometric, weigh the *buffer sizes* equally, which is why native leads there.

TABLE II
DESCRIPTIVE STATISTICS OF *RpS* OBSERVATIONS FOR THE BYPASS *backend* ACROSS DIFFERENT MODELS AND EXECUTION ENVIRONMENTS. ALL VALUES ARE EXPRESSED IN $\mu\text{s/SAMPLE}$.

	Model	Mean	SE
Native	SteerableNAFX	0.390	0.0841
	GuitarLSTM	0.596	0.0628
Chrome	SteerableNAFX	0.338	0.0692
	GuitarLSTM	1.03	0.0175
Firefox	SteerableNAFX	0.340	0.0717
	GuitarLSTM	0.769	0.0352
Safari	SteerableNAFX	0.367	0.0774
	GuitarLSTM	0.782	0.0581

B. Tail Behavior

Table III complements the estimated means with the distribution of the individual measurements for the ONNX Runtime *backend*. Where the mean exceeds the *RTT* (SteerableNAFX at 128 samples in every *environment*, native included, and at 1024 samples on Chrome), every block misses its deadline; everywhere else, misses are confined to *iteration* 0 of a *repetition*, except for five steady-state misses on Chrome (GuitarLSTM,

128 samples), all within six consecutive *iterations* of one *repetition*. Jitter, taken as the steady-state standard deviation, is small relative to the mean: its coefficient of variation is at most 1.5 % in 18 of the 24 configurations, the exceptions all being GuitarLSTM in the browsers (up to 6.3 %, and 18 % on Safari and 16.5 % on Chrome at 128 samples, the latter owing to the misses above). Per block, the spread is 60 to $1060 \mu\text{s}$ in the browsers against 6 to $77 \mu\text{s}$ natively, at most 9.3 % of the block’s budget and well above the timer resolution of Table I. Including the first *iteration* inflates the standard deviation by up to a factor of 6.7 (natively, GuitarLSTM at 128 samples) and the maxima to up to 3.5 times the p99. Bypass runs stay far below the *RTT* even at their maximum ($9.8 \mu\text{s/sample}$). JavaScript-provided *backends* show sporadic misses, largest for JS ONNX on Chrome with GuitarLSTM at 128 samples (1.0 % of blocks); statistics for all 156 configurations are in the supplementary repository [17].

TABLE III
TAIL STATISTICS OF *RpS* FOR THE ONNX RUNTIME *backend* OVER THE 500 MEASUREMENTS OF A RUN, BY *buffer size* (BS). SD, P99, AND MAX IN $\mu\text{s/SAMPLE}$ ($RTT \approx 22.68$); MISS IS THE SHARE OF BLOCKS EXCEEDING THE *RTT*; STARRED COLUMNS EXCLUDE *iteration* 0.

	Model	BS	SD*	p99	Max	Max*	Miss (%)	Miss* (%)
Native	SteerableNAFX	128	0.292	76.0	87.3	77.0	100	100
		1024	0.0328	10.2	10.3	10.3	0	0
		8192	0.00573	1.99	2.00	2.00	0	0
	GuitarLSTM	128	0.0439	4.28	10.5	4.28	0	0
		1024	0.0105	3.35	3.38	3.35	0	0
		8192	0.00938	3.24	3.35	3.27	0	0
Chrome	SteerableNAFX	128	2.10	288	339	291	100	100
		1024	0.346	38.5	43.7	43.7	100	100
		8192	0.0349	7.41	8.68	7.41	0	0
	GuitarLSTM	128	2.09	24.3	36.0	34.9	1.2	1.0
		1024	0.442	12.2	20.0	20.0	0	0
		8192	0.129	11.9	14.2	14.2	0	0
Firefox	SteerableNAFX	128	1.56	143	176	157	100	100
		1024	0.207	18.9	23.3	21.1	0.2	0
		8192	0.0329	3.75	4.43	3.98	0	0
	GuitarLSTM	128	0.468	9.54	29.2	11.7	0.2	0
		1024	0.131	6.66	11.2	7.34	0	0
		8192	0.0504	6.25	7.40	6.49	0	0
Safari	SteerableNAFX	128	0.998	145	195	146	100	100
		1024	0.136	19.6	26.3	19.3	0.2	0
		8192	0.0202	3.83	4.66	3.76	0	0
	GuitarLSTM	128	1.29	14.1	49.5	20.9	0.2	0
		1024	0.201	6.52	14.4	7.68	0	0
		8192	0.0642	5.65	7.87	5.98	0	0

C. Statistical Modeling

Both LMMs yield near-perfect coefficients of determination, with $R^2_{\text{conditional}} = R^2_{\text{marginal}} = 0.999$ for LMM12 and LMM3 alike. As in Schulz and Ackva [12], the random effects (*repetitions*) contribute negligible variance beyond the fixed effects.

For LMM12, all thirty-one terms are significant at $p < 0.001$, including all five main effects. For LMM3, forty-five of sixty-two terms are significant at $p < 0.001$, including all six main effects and the *backend* $\times$ *pre/post-processor* interaction.

The non-significant terms are concentrated among interactions of the *pre/post-processor* with the *iteration number*, so the *pre/post-processor* treatment does not influence how runtime varies across *iterations*. The significant interaction indicates that the effect of the *inference backend* depends on which *pre/post-processor* is in use, and vice versa, so a single overhead estimate averaged over the other factor would be misleading.

1) *Environment Comparison*: Post-hoc estimated marginal means for the *environment* factor, separated by *backend type* and *model*, are shown in Fig. 1. For bypass runs, native execution is the fastest across both *models*. For GuitarLSTM, the *environment* ordering is Native < Safari < Firefox < Chrome, with all pairwise contrasts significant. For SteerableNAFX, absolute differences are smaller and Firefox and Safari trade places, Firefox being faster by a small margin (Firefox/Safari ratio = 0.954), again with all pairwise contrasts significant. When using the ONNX Runtime for inference, Chrome falls even further behind, while Firefox and Safari converge considerably. For SteerableNAFX, their performance is statistically indistinguishable ($p = 0.108$, ratio = 1.009). For GuitarLSTM, Firefox is significantly slower than Safari, though by a small margin. In all cases native execution is significantly faster than any browser. Whether this ranking carries over to other hardware and operating systems is taken up in Section VI.

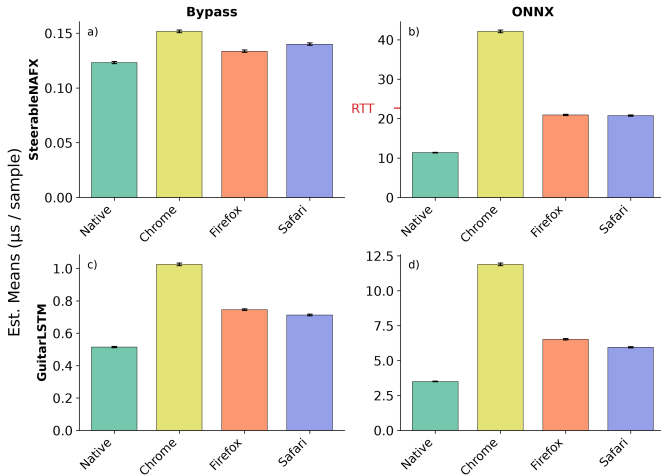


Fig. 1. Estimated geometric means and 95% CI of RpS by execution *environment*, separated by *backend type* (columns) and *model* (rows). a) SteerableNAFX, bypass *backend*; b) SteerableNAFX, ONNX Runtime; c) GuitarLSTM, bypass *backend*; d) GuitarLSTM, ONNX Runtime. The *RTT* is marked as a *y*-axis tick where the axis range reaches it (panel b); estimates are averaged over *buffer sizes*, which Fig. 3 resolves.

2) *Influence of Iteration Number*: Fig. 2 shows estimated marginal means across *iterations* for ONNX Runtime runs. For SteerableNAFX, only Safari shows a significant first-iteration effect, with *iteration 0* elevated above the grand mean; for GuitarLSTM, *iteration 0* is significantly slower than the grand mean in all four *environments*, native execution included, reflecting a first-inference spike. The spike is largest in Safari,

where *iteration 1* also remains significantly elevated. Split by *repetition*, *iteration 0* runs up to 6.9 times above the steady-state mean in a configuration's first *repetition*, right after module instantiation (Safari, GuitarLSTM, 128 samples), but at most 1.43 times in later ones; natively it reaches 2.6 times in the first *repetition* (GuitarLSTM, 128 samples) and at most 5% above in later ones [17].

D. JavaScript Component Overhead

Fig. 3 shows estimated marginal means for all eight *backend* $\times$ *pre/post-processor* (PP) combinations. Because of the significant *backend* $\times$ PP interaction, the two analyses below examine each factor at a fixed level of the other.

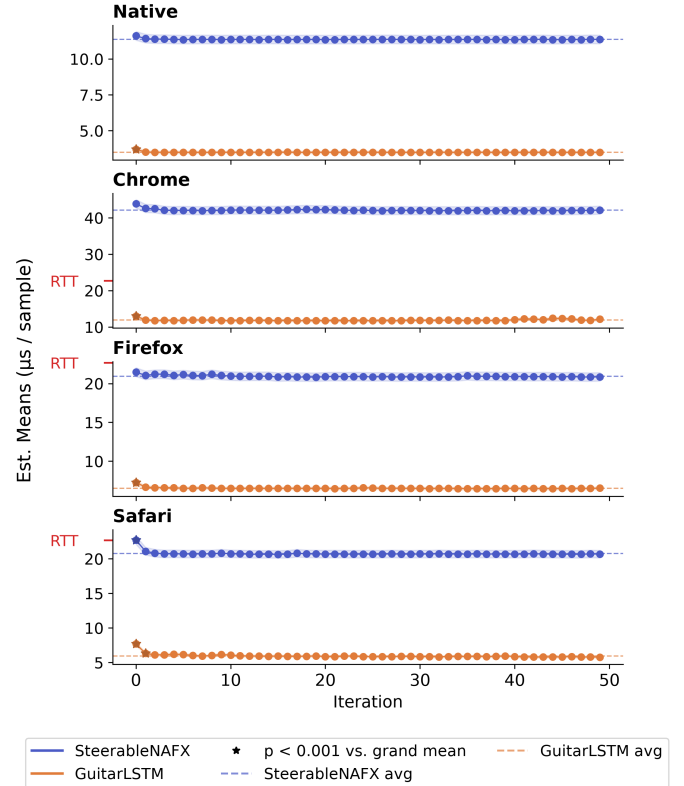


Fig. 2. Estimated geometric means and 95% CI of RpS across *iterations* for the ONNX Runtime *backend*, one subplot per execution *environment*. Dashed lines indicate the per-model grand mean across all *iterations*; starred points (*) mark individual *iterations* significantly different from the grand mean; the *RTT* is marked as a *y*-axis tick where the axis range reaches it.

1) Inference Backend: Bundled vs. JavaScript:

a) *Chrome*: JS ONNX introduces no significant overhead for either *model* ($p > 0.05$). For bypass *backends*, GuitarLSTM likewise shows at most a non-significant trend ($p < 0.05$); for SteerableNAFX the JavaScript bypass *backend* is significantly slower with either *pre/post-processor*, though the effect is negligibly small in magnitude.

b) *Firefox*: For bypass *backends*, the JavaScript *backend* is significantly slower in two conditions, SteerableNAFX with the WebAssembly *pre/post-processor* and GuitarLSTM with the JavaScript one, by less than 5% in both; the other

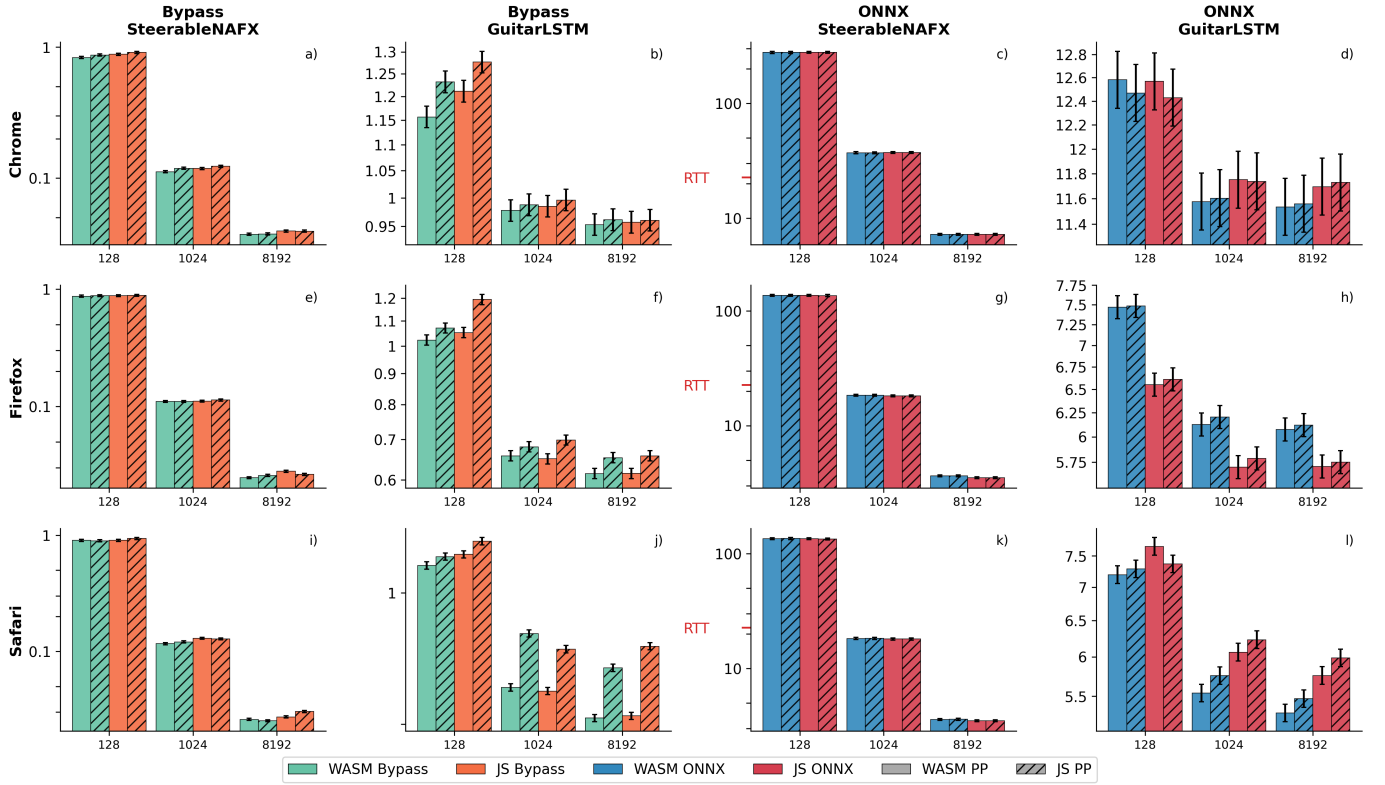


Fig. 3. Estimated geometric means and 95% CI of RpS for the full *backend* $\times$ *pre/post-processor* design. Rows: browser; columns: *backend* group (Bypass left, ONNX right) $\times$ *model*. Bar color indicates the *inference backend* (WebAssembly-bundled vs. JavaScript-provided); bar hatching indicates the *pre/post-processor* (solid: WebAssembly; hatched: JavaScript). *Buffer* sizes 128, 1024, and 8192 are shown on the *x*-axis. The *RTT* is marked as a *y*-axis tick where the axis range reaches it (panels c, g, and k).

two conditions show no significant difference ($p > 0.05$). For ONNX inference, JS ONNX is significantly *faster* for GuitarLSTM and indistinguishable from the WebAssembly runtime for SteerableNAFX.

c) *Safari*: Safari recurrently penalizes JavaScript *backends*. The JavaScript bypass *backend* is significantly slower for SteerableNAFX with either *pre/post-processor*; for GuitarLSTM only a non-significant trend in the same direction is visible with the WebAssembly *pre/post-processor* ($p < 0.05$), which becomes significant when both components are JavaScript-side. With ONNX inference, JS ONNX is significantly slower for GuitarLSTM, though the gap narrows when both the *backend* and *pre/post-processor* are JavaScript-side. SteerableNAFX shows no significant difference between ONNX *backends*.

2) Pre/Post-Processor: Bundled vs. JavaScript:

a) *Chrome*: The JavaScript *pre/post-processor* introduces no meaningful overhead for either *model*: ONNX *backends* show no significant difference ($p > 0.05$), and in bypass conditions only the WebAssembly bypass *backend* reaches significance, for both *models* and at a magnitude that is negligible in absolute terms; with the JavaScript bypass *backend* only non-significant trends remain ($p < 0.01$).

b) *Firefox*: For ONNX inference conditions, no *PP* effect is observed for either *model* ($p > 0.05$). For bypass conditions,

the pattern diverges by *model*: GuitarLSTM shows significant JavaScript *PP* overhead, most pronounced with the JavaScript bypass *backend*; SteerableNAFX shows no significant *PP* effect with either bypass *backend* ($p > 0.01$).

c) *Safari*: For SteerableNAFX, only the JavaScript bypass *backend* condition yields a significant *PP* overhead; all ONNX conditions are non-significant. For GuitarLSTM with bypass *backends*, the JavaScript *pre/post-processor* adds approximately 22 to 25% overhead to RpS . The similar overhead across the WebAssembly and JavaScript bypass *backends* confirms this is a scale effect of *pre/post-processor* cost dominating when inference cost is near zero, rather than a language-boundary penalty per se. With the WebAssembly ONNX Runtime, the JavaScript *PP* adds a small but significant overhead (below 3%). With JS ONNX, no significant *PP* difference is observed ($p = 0.252$).

VI. DISCUSSION

We succeeded in porting Anira to the web, making its native capabilities available to JavaScript developers as the Anira Web npm package. We demonstrate that real-time neural inference is possible in all major web browsers: the benchmarks confirm that Anira Web’s non-inference overhead (its framework together with pre- and post-processing) is negligible relative to the real-time budget across all tested *environments*,

and that inference itself remains within real-time bounds for the models tested, except for SteerableNAFX at the smallest *buffer sizes*, as discussed below. The binding factor is therefore inference cost, which varies considerably across browsers. The environment comparison shows a consistent performance hierarchy. For actual inference, rather than the bypass engine, Chrome runs significantly slower than Firefox and Safari, which converge closely; for SteerableNAFX the two are statistically indistinguishable, and for GuitarLSTM Firefox is significantly slower than Safari. For GuitarLSTM, the estimated means of all three browsers stay within the real-time threshold at every tested *buffer size*. SteerableNAFX, whose much larger receptive field makes each inference considerably more expensive, stays within budget at the larger *buffer sizes* but exceeds a single block’s real-time budget at the smallest *buffer size* (128 samples) in every *environment*, native included, and additionally at 1024 samples on Chrome. Whether a running application would drop audio there, the benchmark does not answer, since it awaits each result before submitting the next block (Section III-E). Anira’s native design targets exactly this situation with a pool of inference threads across which successive blocks are distributed, sustaining throughput even when a single inference exceeds one block’s budget, at the cost of added latency [12]; Anira Web retains the mechanism (Section III-C), but every configuration here ran with one worker, so whether the pool restores real-time throughput in the browser, and at what latency, remains untested. Outside the over-budget configurations, deadline misses are almost entirely confined to the first *iteration* and steady-state jitter stays within 9.3 % of a block’s budget, so steady-state behavior determines feasibility. The cold-start analysis shows that, without warm-up inferences, the first iteration deviates significantly from the grand mean in some configurations, while in others only a non-significant trend is visible. Since WebAssembly compilation, instantiation, worker creation, and session creation precede the timed loop, engine initialization does not enter the measured spike, which instead contains each session’s first inference. Natively the spike is largely confined to a configuration’s first *repetition*, consistent with first-use memory allocation that later sessions reuse; in the browsers it is larger in that first *repetition* after module instantiation (up to 6.9 against 2.6 times the steady-state mean) and remains more visible in later ones, consistent with JavaScript and WebAssembly tiered compilation and call-site optimization adding to the allocation cost. Beyond the first few *iterations*, performance stabilizes in all *environments*. Latency-critical applications should therefore run warm-up blocks through the full pipeline before live audio processing begins. Providing the inference backend and pre- and post-processors via JavaScript, instead of bundling them into the WebAssembly module, carries no significant penalty in most cases. On Chrome and Firefox, the differences between JavaScript-provided and bundled components are mostly non-significant, and where significant they remain within real-time bounds; Firefox even shows a significant advantage for JavaScript-provided ONNX with the GuitarLSTM model. In these *environments* developers can choose freely between bun-

dled WebAssembly components and custom JavaScript implementations, with one caveat: Chrome shows sporadic steady-state outliers at the smallest *buffer size*, with the bundled and the JavaScript-provided ONNX *backend* alike, so for latency-critical use the tail rather than the mean should be consulted there. Safari is the exception: JavaScript-provided components carry a recurring, significant penalty there, indicating higher overhead at the JavaScript–WebAssembly boundary than on Chrome or Firefox. Still, this penalty does not by itself push a configuration beyond the real-time budget.

One possible aspect for future work is GPU-accelerated inference via the WebGPU API [40] or OS-level neural accelerators via WebNN [41], both supported by ONNX Runtime Web but not yet consistently available across browsers without experimental feature flags. The JavaScript backend extension mechanism described in Section III-D provides a natural insertion point for such a backend, although dispatching to those APIs is expected to come with its own overhead. Engines callable from JavaScript, such as TensorFlow.js or ExecuTorch with XNNPACK, could likewise be integrated as further *backend* levels like the JS ONNX *backend* of RQ3, without the from-source static build a bundled engine requires (Section III-B); their in-browser performance remains open. The C API of Section III-D exists only for the Emscripten target; upstream Anira is planned to gain a canonical C API of its own, which the web wrappers would then consume, a migration expected to be a renaming rather than a redesign.

The findings are limited to the two lightweight models tested; how inference cost scales with larger architectures such as transformers remains open. They are measured on a Web Worker rather than the rendering thread (Section IV-B) and with a single inference worker; testing the throughput argument above requires a benchmark that pipelines submissions across several inference workers and reports deadline-miss rate and latency. Nor do they describe end-to-end latency: each measurement covers one block’s round trip through the inference handler, not the buffering, queueing, result reordering, and output delivery around it, and the cost within that round trip is decomposed only into a non-inference share (bypass *backend*) and an inference share (ONNX Runtime *backend*), a finer attribution being deferred since the entire non-inference round trip at 128 samples takes only a few ticks of the browser timers of Table I [17]. Memory consumption is likewise not measured beyond the configured allocation of Section III-B. Finally, all measurements come from a single Apple Silicon machine running macOS, deliberately, since Safari exists only on Apple platforms. The browser ranking of Section V-C1 may therefore not hold on x86 processors, on Windows or Linux, on mobile platforms, or on lower-power devices, whereas the architectural findings, negligible framework overhead and browser-dependent cost of JavaScript-provided components, are properties of the engines and the JavaScript–WebAssembly boundary and are expected, though not verified, to carry over; OS-level scheduling effects, with consequences for the tails of Section V-B, likewise remain open.

ACKNOWLEDGMENT

The AI tool Claude was used for orthography, punctuation, and grammar correction.

REFERENCES

- [1] MDN Contributors. (2025) Web Workers API. Mozilla. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/Worker>
- [2] Mozilla Wiki Contributors. (2013) Audio Data API. Mozilla. [Online]. Available: https://wiki.mozilla.org/Audio_Data_API
- [3] C. Rogers, “Web Audio API,” W3C, W3C Working Draft, Dec. 2011. [Online]. Available: <https://www.w3.org/TR/2011/WD-webaudio-20111215/>
- [4] MDN Contributors. (2025) Web Audio API. Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
- [5] W3C. (2021) W3C recommendation: Web Audio API. W3C. [Online]. Available: <https://www.w3.org/press-releases/2021/webaudio/>
- [6] MDN Contributors. (2025) AudioWorklet - Web APIs. Mozilla. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/AudioWorklet>
- [7] —. (2025) SharedArrayBuffer. Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
- [8] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with WebAssembly,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200.
- [9] Emscripten Contributors. (2025) Emscripten: An LLVM-to-WebAssembly compiler. [Online]. Available: <https://emscripten.org/>
- [10] S. Letz, Y. Orlarey, and D. Foher, “FAUST domain specific audio DSP language compiled to WebAssembly,” in *Companion Proceedings of the The Web Conference 2018*, ser. WWW ’18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, p. 701–709.
- [11] M. Buffa and S. Demont, “Can you DAW it online?: The challenges of a web-based open source workstation,” in *2024 IEEE 5th International Symposium on the Internet of Sounds (IS2)*, 2024, pp. 1–8.
- [12] V. Ackva and F. Schulz, “ANIRA: An architecture for neural network inference in real-time audio applications,” in *2024 IEEE 5th International Symposium on the Internet of Sounds (IS2)*. IEEE, Sep. 2024, p. 1–10.
- [13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: A system for large-scale machine learning,” 2016. [Online]. Available: <https://arxiv.org/abs/1605.08695>
- [14] Microsoft. (2025) ONNX Runtime. [Online]. Available: <https://github.com/microsoft/onnxruntime>
- [15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” 2019. [Online]. Available: <https://arxiv.org/abs/1912.01703>
- [16] A. von Raumer. (2025) ONNXRuntime static WASM builds. [Online]. Available: <https://github.com/Andonvr/ONNXRuntime-Static-WASM-Builds>
- [17] —. (2026) Supplementary material: Benchmark and analysis code. [Online]. Available: <https://github.com/Andonvr/anira-paper-supplementary>
- [18] A. Correy, J. Marcos-Fernández, L. Joglar-Ongay, P. Alonso-Jiménez, X. Serra, and D. Bogdanov, “Audio and music analysis on the web using Essentia.js,” *Transactions of the International Society for Music Information Retrieval*, vol. 4, no. 1, pp. 167–181, 2021.
- [19] M. Borins, “From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten,” in *Proceedings of the Linux Audio Conference (LAC)*, 2014.
- [20] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. B. Viégas, and M. Wattenberg, “TensorFlow.js: Machine learning for the web and beyond,” in *Proceedings of Machine Learning and Systems (MLSys)*, 2019.
- [21] C. F. Ruan, Y. Qin, A. R. Parthasarathy, X. Zhou, R. Lai, H. Jin, Y. Dong, B. Hou, M.-S. Yu, Y. Zhai, S. Agarwal, H. Cao, S. Feng, and T. Chen, “WebLLM: A high-performance in-browser LLM inference engine,” 2024. [Online]. Available: <https://arxiv.org/abs/2412.15803>
- [22] Q. Wang, S. Jiang, Z. Chen, X. Cao, Y. Li, A. Li, Y. Ma, T. Cao, and X. Liu, “Anatomizing deep learning inference in web browsers,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 34, no. 2, pp. 47:1–47:43, 2025.
- [23] F. Jia, S. Jiang, T. Cao, W. Cui, T. Xia, X. Cao, Y. Li, D. Zhang, J. Ren, Y. Liu, L. Qiu, and M. Yang, “Empowering in-browser deep learning inference on edge devices with just-in-time kernel optimizations,” in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2024.
- [24] J. Chowdhury, “RTNeural: Fast neural inferencing for real-time systems,” 2021. [Online]. Available: <https://arxiv.org/abs/2106.03037>
- [25] C. J. Steinmetz and J. D. Reiss, “Efficient neural networks for real-time modeling of analog dynamic range compression,” in *Proceedings of the 152nd Audio Engineering Society Convention (AES Europe 2022)*, 2022. [Online]. Available: <https://arxiv.org/abs/2102.06200>
- [26] A. Wright, E.-P. Damskägg, L. Juvela, and V. Välimäki, “Real-time guitar amplifier emulation with deep learning,” *Applied Sciences*, vol. 10, no. 3, p. 766, 2020.
- [27] A. Caillon and P. Esling, “RAVE: A variational autoencoder for fast and high-quality neural audio synthesis,” 2021. [Online]. Available: <https://arxiv.org/abs/2111.05011>
- [28] (2025) LiteRT issue #2576 comment. [Online]. Available: <https://github.com/google-ai-edge/LiteRT/issues/2576#issuecomment-298934209>
- [29] GitHub. (2025) GitHub Actions. [Online]. Available: <https://github.com/features/actions>
- [30] Emscripten Contributors. (2025) Wasm Audio Worklets API. [Online]. Available: https://emscripten.org/docs/api_reference/wasm_audio_worklets
- [31] —. (2025) Emscripten file system overview. [Online]. Available: https://emscripten.org/docs/porting/files/file_systems_overview
- [32] W3C WebAssembly Working Group, “WebAssembly core specification,” W3C, W3C Candidate Recommendation Draft, 2026. [Online]. Available: <https://www.w3.org/TR/wasm-core-2/>
- [33] ANIRA Contributors. (2025) ANIRA example models. [Online]. Available: <https://github.com/anira-project/example-models>
- [34] B. L. Titzer and J. Sevcik. (2019, 4) A year with Spectre: a V8 perspective. Google. [Online]. Available: <https://v8.dev/blog/spectre>
- [35] P. Grosjean and F. Ibanez, “pastecs: Package for analysis of space-time ecological series,” 2024, in collaboration with M. Etienne. [Online]. Available: <https://github.com/SciViews/pastecs>
- [36] D. Bates, M. Mächler, B. Bolker, and S. Walker, “Fitting linear mixed-effects models using lme4,” *Journal of Statistical Software*, vol. 67, no. 1, p. 1–48, 2015.
- [37] R Core Team, “R: A language and environment for statistical computing,” 2024. [Online]. Available: <https://www.R-project.org/>
- [38] A. Kuznetsova, P. B. Brockhoff, and R. H. B. Christensen, “lmerTest package: Tests in linear mixed effects models,” *Journal of Statistical Software*, vol. 82, no. 13, pp. 1–26, 2017.
- [39] R. V. Lenth, “emmeans: Estimated marginal means, aka least-squares means,” 2025. [Online]. Available: <https://CRAN.R-project.org/package=emmeans>
- [40] MDN Contributors. (2025) WebGPU API. Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API
- [41] W3C Web Machine Learning Working Group, “Web Neural Network API,” W3C, W3C Working Draft, 2025. [Online]. Available: <https://www.w3.org/TR/webnn/>