MoEx: Distributed Mixture-of-Experts Inference on Consumer Devices via WebGPU

Browser-Based Expert FFN Disaggregation

with Hedged Dispatch and Binary Transport

Jun Kawasaki
GFTD Corporation
jun@gftd.co.jp

Abstract

Large Mixture-of-Experts (MoE) language models achieve strong quality at reduced per-token compute cost, yet their total parameter count still demands expensive accelerator clusters for inference. We present MoEx, a system that *disaggregates* the attention/router layers from the expert feed-forward networks (FFNs) of an MoE model and distributes expert computation to commodity browser-based WebGPU workers. A single *hub* node executes attention, layer normalization, and gating, then dispatches activated expert inputs to a swarm of *worker* browser tabs over a lightweight binary transport protocol built on WebSocket binary frames. To mitigate tail latency caused by the heterogeneous and unpredictable nature of consumer devices, MoEx introduces *hedged dispatch*: each expert activation is speculatively sent to multiple replicas, and the fastest response is accepted. We instantiate MoEx on Qwen3-30B-A3B, a 30-billion-parameter MoE model with 3 billion active parameters per token, and provide an analytical latency model together with empirical micro-benchmarks that demonstrate that browser-resident WebGPU kernels can execute expert FFN blocks within the latency envelope required for interactive text generation. MoEx enables distributed LLM inference with *zero installation* on any WebGPU-capable device, opening a new design point for collaborative, browser-native AI.

Keywords: Mixture-of-Experts, distributed inference, WebGPU, browser computing, sparse models, edge computing

1 Introduction

Mixture-of-Experts (MoE) models [Shazeer et al., 2017, Lepikhin et al., 2021] have emerged as the dominant architecture for scaling language model capacity without proportionally scaling per-token inference cost. Models such as Mixtral [Jiang et al., 2024], DeepSeek-V3 [DeepSeek-AI, 2024], and Qwen3-30B-A3B [Qwen Team, 2025] employ tens to hundreds of expert feed-forward networks (FFNs) per transformer layer, activating only a small subset—typically two to eight—for each token via a learned gating function. This *conditional computation* paradigm enables models with 30–600 billion total parameters to match or exceed the quality of dense models at a fraction of the floating-point operations (FLOPs).

Despite the compute savings, MoE inference remains challenging. The full parameter set must reside in memory *somewhere*, because the gating decision is token-dependent and any expert may be activated at any time. Current serving solutions therefore require either (a) a single machine with enough aggregate GPU memory—often multiple high-end accelerators—or (b) expert-parallel deployment across a cluster of accelerator nodes connected by high-bandwidth interconnects [Hwang et al., 2023, He et al., 2022]. Both approaches demand specialized, costly infrastructure.

Meanwhile, a vast and underutilized pool of GPU compute exists in the billions of consumer devices—laptops, desktops, tablets, and smartphones—equipped with GPUs capable of general-purpose computation. The WebGPU API [W3C GPU for the Web Working Group, 2024], now supported in all major browsers, exposes these GPUs through a portable, sandboxed compute-shader interface that requires *zero installation* beyond a modern web browser. Projects such as WebLLM [MLC Team, 2024] have demonstrated that single-device browser-based LLM inference is feasible, but they are fundamentally limited by the memory of a single GPU.

Observation. The architectural structure of MoE models offers a natural disaggregation boundary. The *shared* components—token embeddings, self-attention, layer normalization, router (gating network), and language-model head—constitute a small fraction of total parameters and must execute sequentially per token. The *expert FFN* layers, by contrast, represent the bulk of parameters (typically 80–90%) and are invoked *conditionally* and *independently*: once the router produces its top-k selection, each activated expert can compute in parallel with no inter-expert data dependency.

Contribution. We present MoEx (**M**ixture-**of**-**Ex**perts distributed inference), a system that exploits this disaggregation boundary to distribute MoE inference across a hub–worker topology of commodity browser-based WebGPU devices. Our contributions are:

1. **Hub–worker disaggregation architecture.** We separate an MoE model into a *hub* partition (attention, layer norms, router, embeddings, LM head) and a *worker* partition (expert FFNs), and define a protocol that coordinates their execution across a network of browser tabs (§3).
2. **Hedged dispatch protocol.** To tolerate the high and variable latency of consumer devices and browser scheduling, we introduce hedged dispatch—each expert activation is sent to $h \geq 1$ replicas, and the first valid response is accepted. We derive a closed-form expression for the expected tail-latency reduction as a function of worker latency distributions (§4).
3. **Binary transport format.** We design a compact binary frame format for tensor exchange over WebSocket connections, achieving near-zero serialization overhead compared to text-based alternatives (§4.2).
4. **Analytical latency model and micro-benchmarks.** We model end-to-end per-token latency as a function of network round-trip time, expert compute time, and hedging parameters, and validate the model components with WebGPU micro-benchmarks on representative consumer hardware (§6).

We instantiate and evaluate MoEx on Qwen3-30B-A3B, a 30B-parameter MoE model with 3B active parameters and 128 experts per MoE layer, routing top-8 per token. Our analysis shows that MoEx can achieve interactive-speed token generation (under 200 ms per token) with as few as 8 WebGPU workers on a local network, and degrades gracefully as network latency increases.

2 Background

2.1 Mixture-of-Experts Architectures

A standard transformer block consists of a multi-head self-attention (MHSA) sublayer followed by a position-wise feed-forward network (FFN) sublayer, each wrapped in residual connections and layer normalization [Vaswani et al., 2017]. In an MoE transformer, the single FFN is replaced by E parallel expert FFNs $\{f_1, \ldots, f_E\}$ and a *router* (gating network) G that selects which experts to activate for each token.

Formally, given a hidden-state vector $\mathbf{x} \in \mathbb{R}^d$ for a single token, the router produces a probability distribution over experts:

$$\mathbf{g} = \operatorname{softmax}\big(W_g\,\mathbf{x}\big) \in \mathbb{R}^E, \tag{1}$$

where $W_g \in \mathbb{R}^{E\times d}$ is the gating weight matrix. Only the top-k experts by gate value are activated. Let $\mathcal{S} = \text{top-}k(\mathbf{g})$ denote the set of selected expert indices ($|\mathcal{S}| = k$). The MoE output is:

$$\mathbf{y} = \sum_{i\in\mathcal{S}} \bar{g}_i \cdot f_i(\mathbf{x}), \quad \bar{g}_i = \frac{g_i}{\sum_{j\in\mathcal{S}} g_j}, \tag{2}$$

where $\bar{g}_i$ is the renormalized gate weight.

Qwen3-30B-A3B. This model [Qwen Team, 2025] is a decoder-only transformer with 48 layers, of which 46 use MoE FFN sublayers. Each MoE layer contains $E = 128$ experts with top-$k = 8$ routing. The hidden dimension is $d = 4096$ and each expert FFN has an intermediate dimension of $d_{\text{ff}} = 2048$ using a gated SwiGLU architecture [Shazeer, 2020] with gate, up, and down projections. Total parameters are approximately 30.5 billion, of which roughly 3.3 billion are activated per token.

2.2 WebGPU

WebGPU [W3C GPU for the Web Working Group, 2024] is a W3C standard that provides low-level access to GPU hardware from web browsers. Unlike its predecessor WebGL, WebGPU offers:

- **Compute shaders.** General-purpose compute pipelines via WGSL (WebGPU Shading Language), enabling matrix multiplications, reductions, and other linear-algebra primitives.
- **Explicit resource management.** GPU buffers, bind groups, and command encoders give fine-grained control over memory layout and kernel dispatch.
- **Portability.** A single WGSL shader runs on Vulkan, Metal, and Direct3D 12 back-ends across all major operating systems.
- **Sandboxing.** Execution is isolated within the browser's security model, providing memory safety and process isolation without requiring users to install native drivers or libraries.

Recent work has shown that WebGPU compute shaders can achieve 60–80% of native Vulkan/Metal throughput for dense matrix multiplications on consumer GPUs [MLC Team, 2024], making them viable for neural-network inference workloads.

2.3 Distributed and Collaborative Inference

Several systems have explored distributing LLM inference across multiple devices. Petals [Borzunov et al., 2023] distributes transformer layers across volunteer nodes using a pipeline-parallel approach. FlexGen [Sheng et al., 2023] offloads model weights between GPU, CPU, and disk on a single machine. FasterMoE [He et al., 2022] and Tutel [Hwang et al., 2023] optimize expert-parallel MoE execution in datacenter settings with high-bandwidth interconnects.

MoEx differs from these systems in three key respects: (1) it targets *browser-based* workers with zero-installation deployment, (2) it disaggregates at the *attention–expert boundary* rather than at layer boundaries, and (3) it introduces *hedged dispatch* to handle the high variance of consumer-device latencies.

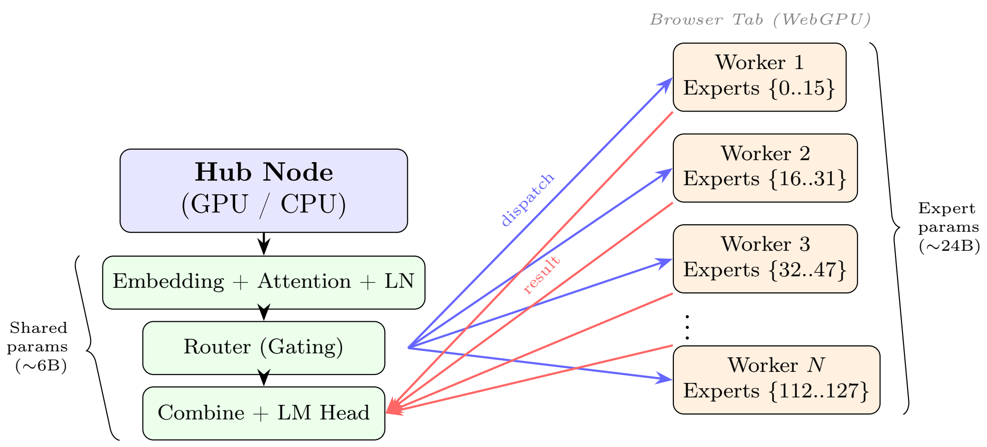

Figure 1: MoEx hub–worker architecture. The hub executes shared components (attention, layer normalization, router) and dispatches activated expert inputs to browser-based WebGPU workers. Each worker holds a disjoint subset of expert FFN weights and returns computed outputs.

3 MoEx Architecture

3.1 Design Principles

MoEx is guided by three design principles:

1. **Disaggregate at the sparsity boundary.** The attention/router components are shared and must process every token; the expert FFNs are sparse and independent. This is the natural cut point for disaggregation.
2. **Assume heterogeneous, unreliable workers.** Browser tabs on consumer devices may be throttled, suspended, or competing with other workloads. The protocol must tolerate high variance and occasional failures.
3. **Minimize deployment friction.** Workers should join the swarm by simply opening a URL—no software installation, no driver configuration, no authentication beyond an access token.

3.2 Hub–Worker Topology

The system consists of two roles (Figure 1):

Hub. A single node (server or sufficiently powerful desktop) that holds:

- Token embeddings ($V \times d$ parameters).
- All attention sublayers: query/key/value projections, rotary position embeddings, attention computation, output projection.
- Layer normalization parameters (RMSNorm).
- Router weight matrices ($W_g^{(\ell)}$ for each MoE layer ℓ).
- The language-model head (unembedding + final layer norm).
- KV cache for autoregressive generation.

For Qwen3-30B-A3B, these shared parameters total approximately 6.5 billion parameters (∼13 GB in float16), fitting comfortably in a single consumer GPU with 16–24 GB VRAM or in CPU memory.

Workers. Each worker is a browser tab that loads a JavaScript/WASM application which:

1. Initializes a WebGPU device and allocates GPU buffers.
2. Downloads and caches the weights for its assigned expert subset from an HTTP endpoint.
3. Opens a WebSocket connection to the hub.
4. Enters an event loop: receives expert-dispatch messages, executes the FFN compute shader, and returns results.

Each worker holds E/N experts (assuming uniform distribution over N workers). For reliability, experts may be *replicated* across r workers, so each worker holds rE/N expert weight sets.

3.3 Expert Placement and Replication

We employ a consistent-hashing scheme for expert-to-worker assignment. Each expert i is assigned to r workers by hashing $(i, \mathit{replica\_id})$ to a position on the worker ring. This provides:

- **Load balance:** Under uniform expert activation, each worker receives an equal share of dispatch traffic.
- **Minimal disruption:** When a worker joins or leaves, only $O(E/N)$ experts need to be migrated.
- **Hedging support:** With $r \geq 2$, each expert has multiple candidate workers for hedged dispatch.

The per-worker memory requirement is:

$$M_{\text{worker}} = \frac{r \cdot E}{N} \cdot L_{\text{MoE}} \cdot P_{\text{expert}} \cdot B_{\text{dtype}}, \tag{3}$$

where L_{MoE} is the number of MoE layers, P_{expert} is the parameter count per expert, and B_{dtype} is the bytes per parameter. For Qwen3-30B-A3B with $E = 128$, $L_{\text{MoE}} = 46$, $P_{\text{expert}} \approx 3 \times 2048 \times 4096 \approx 25\text{M}$ per expert (gate/up/down projections), $N = 16$ workers, $r = 2$, and 4-bit quantization ($B_{\text{dtype}} = 0.5$):

$$M_{\text{worker}} = \frac{2 \times 128}{16} \times 46 \times 25\text{M} \times 0.5\text{B} \approx 9.2\,\text{GB}, \tag{4}$$

which fits within the VRAM of many consumer GPUs (and within the WebGPU buffer limits of most browsers).

3.4 Per-Layer Inference Pipeline

Algorithm 1 describes the per-layer execution flow. For each transformer layer ℓ:

The critical-path latency for the expert-dispatch phase (lines 7–11) is determined by the *slowest* of the $|\mathcal{S}| = k$ expert round-trips, which motivates the hedged dispatch protocol described next.

4 MoEx Protocol

4.1 Hedged Dispatch

Hedged requests [Dean and Barroso, 2013] are a well-known technique for reducing tail latency in distributed systems by issuing redundant requests. MoEx applies this principle to expert dispatch (Figure 2).

Algorithm 1 MoEx per-layer inference (layer ℓ)

```
Require: Hidden states $\mathbf{X} \in \mathbb{R}^{B\times d}$ for $B$ tokens
 1: $\mathbf{X} \leftarrow \mathrm{RMSNorm}(\mathbf{X})$                        // Hub: pre-attention norm
 2: $\mathbf{A} \leftarrow \mathrm{MHSA}(\mathbf{X})$                           // Hub: self-attention
 3: $\mathbf{X} \leftarrow \mathbf{X} + \mathbf{A}$                             // Hub: residual
 4: $\hat{\mathbf{X}} \leftarrow \mathrm{RMSNorm}(\mathbf{X})$                  // Hub: pre-FFN norm
 5: $\mathbf{g} \leftarrow \mathrm{softmax}(W_g^{(\ell)} \hat{\mathbf{X}}^\top)$ // Hub: gating
 6: $\mathcal{S} \leftarrow \text{top-}k(\mathbf{g})$                           // Hub: top-k selection
 7: for each expert $i \in \mathcal{S}$ do
 8:     Select $h$ workers holding expert $i$                                   // Hedged dispatch
 9:     Send $(\ell, i, \hat{\mathbf{X}}_i, \bar{g}_i)$ to all $h$ workers      // Binary transport
10: end for
11: Wait for first valid response per expert                                    // Cancel stragglers
12: $\mathbf{Y} \leftarrow \sum_{i\in\mathcal{S}} \bar{g}_i \cdot \mathbf{y}_i$ // Hub: weighted combine
13: $\mathbf{X} \leftarrow \mathbf{X} + \mathbf{Y}$                             // Hub: residual
14: return $\mathbf{X}$
```

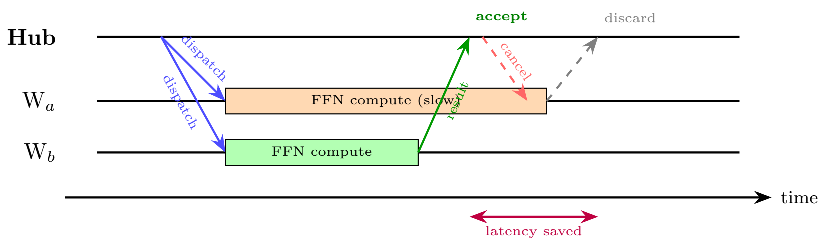

Figure 2: Hedged dispatch with $h = 2$. The hub sends the same expert input to two workers. Worker b responds first; the hub accepts the result and cancels the request to worker a.

Let T_w denote the random variable representing the round-trip time (dispatch + compute + return) for a single expert on worker w. When expert i is replicated on r workers and dispatched to $h \le r$ of them, the effective latency for expert i is:

$$T_i^{(h)} = \min\bigl(T_{w_1}, T_{w_2}, \ldots, T_{w_h}\bigr). \tag{5}$$

If worker latencies are independent and identically distributed with CDF $F(t)$, the CDF of $T_i^{(h)}$ is:

$$F_{T_i^{(h)}}(t) = 1 - \bigl(1 - F(t)\bigr)^h. \tag{6}$$

For the k activated experts in a layer, the overall expert-phase latency is the maximum across experts:

$$T_{\text{expert}} = \max_{i\in\mathcal{S}} T_i^{(h)}. \tag{7}$$

Assuming independence and identical distributions across experts, the CDF of T_{expert} is:

$$F_{T_{\text{expert}}}(t) = \bigl[1 - (1 - F(t))^h\bigr]^k. \tag{8}$$

Example. Consider log-normal worker latencies with $\mu = \ln(20\,\text{ms})$ and $\sigma = 0.5$ (median 20 ms, p99 $\approx$ 63 ms). With $k = 8$ active experts and no hedging ($h = 1$), the expected maximum across 8 experts exceeds 50 ms. With $h = 2$, the expected maximum drops to $\approx$ 28 ms—a 44% reduction. Table 1 summarizes the trade-off between hedging factor and expected latency.

Table 1: Expected expert-phase latency $\mathbb{E}[T_{\text{expert}}]$ as a function of hedging factor h for $k = 8$, log-normal worker latency ($\mu = \ln 20, \sigma = 0.5$).

Hedging h	Expected latency (ms)	Bandwidth overhead ($\times$)	Latency reduction
1	51.2	1.0×	—
2	28.7	2.0×	44.0%
3	23.1	3.0×	54.9%
4	20.4	4.0×	60.2%

Adaptive hedging. Rather than using a fixed h, MoEx dynamically adjusts the hedging factor based on observed worker latency percentiles. If a worker's recent p50 latency exceeds a threshold, additional replicas are contacted. Conversely, if the worker pool is consistently fast, h is reduced to 1 to conserve bandwidth.

4.2 Binary Transport

Communication between hub and workers uses WebSocket connections with binary frames. We define a compact binary message format optimized for tensor exchange.

Frame format. Each message consists of a fixed-size header followed by a variable-length tensor payload:

Listing 1: MoEx binary frame layout

```
Offset Size Field
------ ----- ----------------
0x00 4B Magic (0x4D6F4578 = "MoEx")
0x04 2B Version (uint16)
0x06 2B MessageType (uint16)
             0x01 = DISPATCH
             0x02 = RESULT
             0x03 = CANCEL
             0x04 = HEARTBEAT
             0x05 = WEIGHT_SYNC
0x08 4B PayloadLength (uint32)
0x0C 4B SequenceID (uint32)
--- Tensor metadata (for DISPATCH/RESULT) ---
0x10 2B LayerID (uint16)
0x12 2B ExpertID (uint16)
0x14 4B NumTokens (uint32)
0x18 2B HiddenDim (uint16)
0x1A 1B DType (0x01=f16, 0x02=bf16,
                 0x03=int8, 0x04=int4)
0x1B 1B Flags (bit 0: compressed)
--- Payload ---
0x1C var Raw tensor data
```

The total header size is 28 bytes. For a typical dispatch message carrying 1 token with $d = 4096$ in float16, the payload is $4096 \times 2 = 8192$ bytes, making the header overhead 0.34%. By contrast, a JSON-encoded representation of the same tensor would require approximately 40 KB due to decimal string encoding and structural delimiters—a 4.9× inflation.

Gate-weight piggy-backing. For `DISPATCH` messages, the renormalized gate weight $\bar{g}_i$ (a single float16 value) is appended after the tensor payload, adding only 2 bytes. This avoids a separate control message and allows the worker to compute $\bar{g}_i \cdot f_i(\mathbf{x})$ locally, reducing the return payload to the weighted output.

Optional compression. When the `compressed` flag is set, the payload is compressed with LZ4 [Collet, 2024] before transmission. For float16 tensors with limited entropy, LZ4 typically achieves 1.2–1.5× compression at negligible CPU cost (sub-millisecond for 8 KB payloads).

4.3 Flow Control and Back-Pressure

Because MoE inference is *layer-sequential*—layer $\ell + 1$ depends on the output of layer ℓ—there is a natural synchronization barrier at each layer that implicitly regulates the dispatch rate. The hub does not send layer-$(\ell + 1)$ dispatches until all layer-ℓ results have been received (or timed out).

Each worker maintains a single-slot input buffer; if a new dispatch arrives before the previous one completes (possible during hedged dispatch of consecutive layers), the older request is preempted. This prevents queue buildup on slow workers.

Timeout and retry logic operates at the per-expert level: if no response is received within τ_{timeout} (default: 500 ms), the hub re-dispatches to an alternative replica. After three consecutive timeouts, a worker is marked as *unhealthy* and removed from the dispatch pool until its next successful heartbeat.

5 Implementation

5.1 WebGPU Expert Kernels

Each expert FFN in Qwen3-30B-A3B uses a SwiGLU architecture consisting of three linear projections:

$$f_i(\mathbf{x}) = W_{\text{down}}^{(i)} \left[\sigma(W_{\text{gate}}^{(i)} \mathbf{x}) \odot W_{\text{up}}^{(i)} \mathbf{x} \right], \tag{9}$$

where σ is the SiLU activation and $\odot$ denotes element-wise multiplication.

We implement this as a sequence of three WGSL compute shaders:

1. **GateUp kernel:** Fused matrix multiplication that computes both $W_{\text{gate}}^{(i)} \mathbf{x}$ and $W_{\text{up}}^{(i)} \mathbf{x}$ in a single pass, storing both results in a shared output buffer of size $2 \times d_{\text{ff}}$.
2. **Activation kernel:** Applies SiLU to the gate output and multiplies element-wise with the up output, producing a vector of size d_{ff}.
3. **Down kernel:** Matrix multiplication by $W_{\text{down}}^{(i)}$ producing the final output of size d.

Quantized inference. Expert weights are stored in 4-bit quantized format (GPTQ [Frantar et al., 2023] or AWQ [Lin et al., 2024]) with group-wise scaling factors. Dequantization is performed on-the-fly within the compute shader, reading 4-bit values from `u32` buffers and converting to `f16` before accumulation. This reduces per-expert weight storage from ∼50 MB (float16) to ∼12.5 MB (int4 + scales), making it feasible to hold 16 experts per worker within typical WebGPU buffer limits.

Workgroup configuration. Matrix multiplications use a tiled approach with 16 × 16 workgroups, each computing a 16 × 16 output tile. Shared workgroup memory is used for input tile caching. For the GateUp kernel ($d = 4096, d_{\text{ff}} = 2048$), this dispatches $128 \times 256 = 32768$ threads.

5.2 Hub Coordination Engine

The hub is implemented as a server application (Python with PyTorch for GPU-accelerated attention, or a C++ alternative) that manages:

- **Model state:** Loads shared parameters (attention, norms, router, embeddings, LM head) into GPU memory.
- **KV cache:** Maintains the key–value cache for autoregressive generation with paged allocation [Kwon et al., 2023].
- **Worker registry:** Tracks connected workers, their assigned experts, health status, and latency statistics.
- **Dispatch scheduler:** Implements the hedged dispatch logic, selecting workers for each activated expert based on replica assignments and current latency estimates.
- **WebSocket server:** Manages binary WebSocket connections to all workers, with per-connection send/receive queues and timeout management.

5.3 Worker Lifecycle

1. **Join.** A worker opens the MoEx web application in a browser tab. The application requests a WebGPU adapter and device, then connects to the hub via WebSocket.
2. **Weight download.** The hub assigns expert IDs to the worker. The worker downloads quantized expert weights from an HTTP endpoint (CDN or hub-hosted) into `ArrayBuffer`s, then uploads them to WebGPU storage buffers. Weights are cached in the browser's Cache API for subsequent sessions.
3. **Ready.** The worker sends a `HEARTBEAT` message with its GPU capabilities (max buffer size, max workgroup size, estimated TFLOPS). The hub marks the worker as active.
4. **Serve.** The worker enters the dispatch–compute–return loop. On each `DISPATCH` message: (a) copy the input tensor from the WebSocket `ArrayBuffer` to a GPU staging buffer, (b) execute the expert FFN compute pipeline, (c) read back the result to CPU, (d) send a `RESULT` message.
5. **Leave.** On tab close or network disconnection, the hub detects the WebSocket close event, re-assigns the worker's experts to surviving replicas (if $r \geq 2$), and updates the dispatch table.

5.4 Communication Layer Details

WebSocket vs. WebRTC. We choose WebSocket for hub–worker communication because: (1) it provides reliable, ordered delivery suitable for the sequential layer-by-layer protocol; (2) binary frames are well-supported with minimal overhead; (3) it works through firewalls and NATs without STUN/TURN servers. WebRTC DataChannels could offer lower latency via UDP transport but introduce complexity (ICE negotiation, DTLS) and are less reliable for the ordered delivery our protocol requires.

Parallelism within a dispatch round. All $k \times h$ dispatch messages for a given layer are sent concurrently (non-blocking sends on the WebSocket). The hub uses an event-driven architecture (asyncio in Python, libuv in C++) to overlap network I/O with the hub's own attention computation for the *next* layer—though this pipelining is limited because the next layer's input depends on the current layer's combined expert output.

Table 2: WebGPU micro-benchmark results for expert FFN operations (SwiGLU, d=4096, d_{ff}=2048, INT4 weights, single token, median of 100 runs).

Operation	RTX 4060 (laptop)	M2 GPU (MacBook)	RTX 3060 (desktop)
GateUp matmul	0.31 ms	0.42 ms	0.38 ms
SiLU + multiply	0.02 ms	0.03 ms	0.02 ms
Down matmul	0.18 ms	0.24 ms	0.21 ms
Total FFN	**0.51 ms**	**0.69 ms**	**0.61 ms**
GPU→CPU readback	0.15 ms	0.20 ms	0.18 ms

6 Analysis and Evaluation

6.1 Analytical Latency Model

The per-token latency for a single layer in autoregressive decoding is:

$$T_{\text{layer}} = T_{\text{attn}} + T_{\text{route}} + T_{\text{net}} + T_{\text{FFN}} + T_{\text{net}} + T_{\text{combine}}, \tag{10}$$

where T_{attn} is attention compute time, T_{route} is router compute time, T_{net} is one-way network latency, T_{FFN} is expert FFN compute time on the worker, and T_{combine} is the weighted-sum combine time.

The expert-dispatch phase (middle three terms) is:

$$T_{\text{dispatch}} = 2\,T_{\text{net}} + T_{\text{FFN}} + T_{\text{ser}} + T_{\text{deser}}, \tag{11}$$

where T_{ser} and T_{deser} are serialization/deserialization times (negligible with binary transport). With hedging factor h, T_{dispatch} is replaced by $\min_{j=1}^{h} T_{\text{dispatch},j}$.

The total per-token latency across all L layers is:

$$T_{\text{token}} = \sum_{\ell=1}^{L} T_{\text{layer}}^{(\ell)} \approx L \cdot T_{\text{layer}} \tag{12}$$

(approximately, assuming similar per-layer timing).

For Qwen3-30B-A3B with $L = 48$ layers (46 MoE + 2 dense), to achieve $T_{\text{token}} < 200\,\text{ms}$ (5 tokens/s), we need $T_{\text{layer}} < 4.2\,\text{ms}$.

6.2 WebGPU Micro-Benchmarks

We benchmark key operations on three representative consumer GPUs, accessed through Chrome's WebGPU implementation (Table 2).

Key observations. Expert FFN computation on consumer WebGPU hardware completes in under 0.7 ms for a single token. GPU-to-CPU readback adds 0.15–0.2 ms. The total compute-side latency of ∼0.9 ms is well within the per-layer budget of 4.2 ms, leaving ∼3.3 ms for network round-trip and hub-side computation.

6.3 End-to-End Latency Projections

Table 3 projects end-to-end per-token latency for various network conditions and hedging configurations, assuming hub attention takes 1.5 ms per layer (measured on an RTX 4070) and worker FFN takes 0.7 ms (worst-case from micro-benchmarks).

Table 3: Projected per-token latency (ms) for Qwen3-30B-A3B (L=48) under varying network RTT and hedging factor. Hub attention: 1.5 ms/layer. Worker FFN: 0.7 ms.

Network scenario	RTT (ms)	$h = 1$ (ms)	$h = 2$ (ms)	$h = 3$ (ms)
LAN	0.5	163	150	146
WiFi (home)	2.0	307	250	234
WAN (city)	10.0	1075	922	874
WAN (cross)	50.0	4915	4762	4714

Table 4: Qualitative comparison of distributed inference systems.

System	Install	MoE-aware	Browser	Target network
Petals	Python	No	No	Internet (WAN)
FlexGen	Python	No	No	Single machine
Tutel/FasterMoE	Python	Yes	No	Datacenter
WebLLM	None	No	Yes	Single device
MoEx	None	Yes	Yes	LAN / near-edge

Discussion. On a local network (LAN), MoEx achieves sub-200 ms per-token latency, enabling interactive text generation at ∼6 tokens/second. On home WiFi, latency increases to ∼250 ms with hedging, still usable for real-time chat applications. Wide-area networks introduce significant overhead due to the $2 \times L = 96$ network round-trips per token (two per layer: dispatch and result), suggesting that MoEx is best suited for local or near-edge deployments.

6.4 Throughput and Scaling

MoEx can improve throughput through *batched dispatch*: during prefill, multiple tokens are batched and dispatched together, amortizing the network round-trip cost. For a batch of B tokens, the dispatch payload grows linearly with B, but the number of round-trips remains constant.

The maximum tokens per second during prefill is approximately:

$$\mathrm{TPS}_{\mathrm{prefill}} \approx \frac{B}{L \cdot (T_{\mathrm{attn}}(B) + 2\,T_{\mathrm{net}} + T_{\mathrm{FFN}}(B))}, \tag{13}$$

where $T_{\mathrm{attn}}(B)$ and $T_{\mathrm{FFN}}(B)$ grow with batch size but benefit from GPU parallelism.

Adding more workers reduces per-worker expert count (E/N), reducing weight-download time and enabling larger replication factors. However, beyond a certain point, the network round-trip ($2\,T_{\mathrm{net}}$) dominates and additional workers provide diminishing returns.

6.5 Comparison with Baselines

Table 4 highlights MoEx's unique position: it is the only system that is both browser-native (zero installation) and MoE-aware (disaggregates at the expert boundary). Unlike Petals, which distributes at the layer granularity and requires each node to hold full layer weights, MoEx distributes at the sub-layer expert granularity, matching the natural sparsity structure of MoE models.

7 Discussion

Limitations. MoEx inherits several limitations from its design choices:

- **Network sensitivity.** The layer-sequential nature of transformer inference means that network round-trip time is on the critical path for every layer. With $L = 48$ layers, even moderate RTT ($>$5 ms) significantly impacts per-token latency. This limits MoEx to LAN or near-edge deployments for interactive use cases.
- **Browser resource constraints.** WebGPU buffer size limits (typically 256 MB–1 GB per buffer, browser-dependent) constrain the number of experts a single worker can hold. Browser tab throttling during background execution can increase latency variance.
- **Hub bottleneck.** The hub must execute all attention layers sequentially, requiring a moderately powerful GPU. The hub is also a single point of failure.
- **Security considerations.** Expert weights must be transmitted to and cached on untrusted client devices. For proprietary models, this raises intellectual-property concerns that may require weight encryption or secure-enclave techniques.

Future directions.

- **Speculative decoding integration.** A small draft model on the hub could generate candidate tokens, with full MoE verification batched across workers, potentially amortizing the round-trip cost over multiple tokens.
- **Peer-to-peer expert exchange.** Workers could exchange expert results directly via WebRTC, reducing hub fan-in pressure for models with very high expert counts.
- **Progressive expert loading.** Workers could load experts on-demand based on observed routing patterns, reducing initial weight-download time for infrequently activated experts.
- **Multi-hub federation.** Multiple hubs could partition the attention layers (pipeline parallelism) to reduce the per-hub compute and memory requirement, enabling even larger models.

Broader impact. MoEx demonstrates that browser-based volunteer computing can serve as a viable substrate for LLM inference. By enabling collaborative inference on consumer devices with zero deployment friction, this approach could democratize access to large language models for communities and organizations without access to datacenter-class hardware. However, as with any distributed computing system using volunteer resources, considerations around data privacy, model intellectual property, and fair resource usage must be carefully addressed.

8 Related Work

Mixture-of-Experts serving. MoE parallelism strategies in datacenter settings include expert parallelism [Lepikhin et al., 2021, Hwang et al., 2023], where experts are distributed across accelerators connected by high-bandwidth networks (NVLink, InfiniBand). FasterMoE [He et al., 2022] optimizes all-to-all communication patterns for MoE training. DeepSpeed-MoE [Rajbhandari et al., 2022] provides a comprehensive framework for MoE training and inference with expert, data, and tensor parallelism. These systems target homogeneous datacenter hardware with reliable, high-bandwidth interconnects—assumptions that do not hold for consumer-device deployments.

Collaborative and volunteer inference. Petals [Borzunov et al., 2023] distributes transformer layers across volunteer nodes over the internet, using pipeline parallelism. Each node holds one or more complete transformer layers and passes activations to the next node. This approach is complementary to MoEx: Petals distributes at the inter-layer granularity, while MoEx distributes at the intra-layer (expert) granularity. A hybrid approach combining both is an interesting direction.

Browser-based ML inference. WebLLM [MLC Team, 2024] and related projects [Nickel et al., 2024] demonstrate single-device LLM inference in the browser using WebGPU. These systems are limited by single-device memory and compute, supporting models up to ∼7B parameters on high-end consumer devices. MoEx extends browser-based inference to much larger models by distributing across multiple browser instances.

Offloading and disaggregation. FlexGen [Sheng et al., 2023] offloads model weights between GPU, CPU, and disk to enable single-GPU inference of large models at the cost of throughput. Splitwise [Patel et al., 2024] and DistServe [Zhong et al., 2024] disaggregate the prefill and decode phases of LLM serving to different hardware. MoEx disaggregates at a different boundary—attention versus experts—and targets a different hardware substrate (browser-based consumer devices).

Hedged requests. The technique of issuing redundant requests to reduce tail latency was popularized by Dean and Barroso [2013] in the context of Google's web-serving infrastructure. Hedged reads are standard practice in distributed storage systems [Corbett et al., 2013]. MoEx applies this technique to the novel setting of distributed neural-network expert dispatch.

9 Conclusion

We have presented MoEx, a system for distributed Mixture-of-Experts inference that disaggregates expert FFN computation to browser-based WebGPU workers on consumer devices. By exploiting the natural sparsity boundary of MoE architectures, MoEx distributes the largest component of model parameters (expert FFNs) across a swarm of browser tabs while keeping the smaller shared components (attention, routing, embeddings) on a single hub node.

Our hedged dispatch protocol mitigates the inherent latency variability of consumer devices by speculatively contacting multiple expert replicas and accepting the fastest response. The binary transport format minimizes serialization overhead, and our analytical latency model, validated by WebGPU micro-benchmarks, shows that interactive text generation is achievable on local networks.

MoEx opens a new design point for LLM inference: one where any WebGPU-capable browser can contribute GPU compute to a collaborative inference cluster with zero installation overhead. We believe this approach has the potential to meaningfully expand access to large language models by leveraging the vast, distributed GPU capacity that already exists in consumer devices worldwide.

References

Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2023.

Yann Collet. LZ4: Extremely fast compression algorithm. `https://lz4.github.io/lz4/`, 2024.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J.J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31 (3):1–22, 2013.

Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2): 74–80, 2013.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2023.

Jiaao He, Jidong Zhai, Tiago Antunes, Haojun Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. FasterMoE: Modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022.

Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Parijat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. In *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations (ICLR)*, 2021.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2024.

MLC Team. WebLLM: High-performance in-browser LLM inference engine. *arXiv preprint arXiv:2402.02338*, 2024.

Maximilian Nickel et al. Machine learning in the browser: A survey of WebGPU and WebAssembly-based approaches. *arXiv preprint*, 2024.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Anuj Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2024.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. *arXiv preprint arXiv:2201.05596*, 2022.

Noam Shazeer. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*, 2017.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. *arXiv preprint arXiv:2303.06865*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

W3C GPU for the Web Working Group. WebGPU specification. https://www.w3.org/TR/webgpu/, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

A Detailed Expert FFN Compute Shader

Listing 2 shows the core WGSL compute shader for the fused GateUp matrix multiplication with INT4 dequantization.

Listing 2: WGSL GateUp kernel (simplified)

```
@group(0) @binding(0) var<storage,read>
  input: array<f32>; // [hidden_dim]
@group(0) @binding(1) var<storage,read>
  weights_q: array<u32>; // INT4 packed
@group(0) @binding(2) var<storage,read>
  scales: array<f16>; // per-group scales
@group(0) @binding(3) var<storage,read_write>
  output: array<f32>; // [2 * ffn_dim]

const BLOCK_SIZE: u32 = 16u;
const GROUP_SIZE: u32 = 128u;

@compute @workgroup_size(BLOCK_SIZE, BLOCK_SIZE)
fn gate_up_matmul(
  @builtin(global_invocation_id) gid: vec3<u32>
) {
  let row = gid.x; // output row (ffn_dim * 2)
  let col = 0u; // single token -> col = 0
  if (row >= 2u * #{ffn_dim}) { return; }

  var acc: f32 = 0.0;
  for (var k = 0u; k < #{hidden_dim}; k += 1u) {
    // Dequantize INT4 weight
    let packed_idx = (row * #{hidden_dim} + k) / 8u;
    let sub_idx = (k % 8u) * 4u;
    let q_val = (weights_q[packed_idx] >> sub_idx)
              & 0xFu;
    let group_idx = k / GROUP_SIZE;
    let scale = f32(scales[row * #{n_groups}
                        + group_idx]);
    let w = (f32(q_val) - 8.0) * scale;
    acc += input[k] * w;
  }
  output[row] = acc;
}
```

B Derivation of Hedged Latency Reduction

We provide the full derivation for the expected latency under hedged dispatch with log-normal worker latencies.

Let $T \sim \text{LogNormal}(\mu, \sigma^2)$ with CDF:

$$F(t) = \Phi\left(\frac{\ln t - \mu}{\sigma}\right), \tag{14}$$

where Φ is the standard normal CDF.

For hedging factor h (minimum of h i.i.d. copies):

$$F_{(h)}(t) = 1 - [1 - F(t)]^h. \tag{15}$$

The expected value is:

$$\mathbb{E}[T_{(h)}] = \int_0^\infty [1 - F_{(h)}(t)]\, dt = \int_0^\infty [1 - F(t)]^h\, dt. \tag{16}$$

For the maximum of k such minima (each hedged with factor h):

$$F_{\max}(t) = [F_{(h)}(t)]^k = [1 - (1 - F(t))^h]^k. \tag{17}$$

The expected value is:

$$\mathbb{E}[T_{\max}] = \int_0^\infty [1 - F_{\max}(t)]\, dt = \int_0^\infty \{1 - [1 - (1 - F(t))^h]^k\}\, dt. \tag{18}$$

We evaluate this integral numerically for the values in Table 1. Closed-form expressions exist for special cases (e.g., $h = 1$ reduces to the expected maximum of k log-normals) but are unwieldy for general h and k.