ChatGPT

WebLLM configuration guide for backend engineers

1. Overview of WebLLM

WebLLM is a **browser-based inference engine** for large language models (LLMs). It runs models directly on the user's GPU via WebGPU and provides an **OpenAI-compatible chat API**, including support for streaming responses, JSON-mode and function-calling [1]. Instead of sending data to a remote API, WebLLM lets you ship a model to the browser, preserving **privacy and reducing server costs** [2]. The engine comes with a set of pre-converted models (Llama, Phi, Gemma, RedPajama, Mistral, Qwen, etc.) [3] and exposes them through a configuration object called `prebuiltAppConfig.model_list` [4]. Backend engineers can select a model from this list based on the use-case (e.g., model size, instruction tuning, quantisation) and then wire it into their application.

2. Locating and understanding the model list

WebLLM registers each available model as a `ModelRecord`. These records reside in `webllm.prebuiltAppConfig.model_list` [4]. Each record contains:

- `model_id` – unique identifier used by the engine (e.g., `"Llama-3.1-8B-Instruct"`).
- **Human-readable name / family** – identifies the architecture (Llama, Mistral, etc.) [3].
- **Quantisation / memory requirements** – denotes whether the weights use 4-bit or 8-bit quantisation. Models with lower quantisation require less download time but may sacrifice accuracy.
- **Additional metadata** – version, languages and capability notes (e.g., instruct-tuned models are optimized for following instructions, which is beneficial for parsing tasks).

To view the available models at runtime, you can inspect this list:

```
import * as webllm from "@mlc-ai/web-llm";

// Map to a list of model_ids
document.addEventListener("DOMContentLoaded", () => {
  const availableModels = webllm.prebuiltAppConfig.model_list.map((m) =>
m.model_id);
  console.log("Supported model IDs:", availableModels);
});
```

> **Selecting a model:** For analysing real-estate listings, choose an **instruction-tuned model** (e.g., `Llama-3.x-Instruct`, `Mistral-7B-Instruct` or `Gemma-Instruct`) because these models are trained to follow structured prompts and produce coherent outputs. Smaller

models like Phi-2 can run on devices with limited GPU memory but may produce less detailed analysis.

3. Installing WebLLM

Install WebLLM as an NPM dependency in your frontend or extension package. Since the backend code will load the model in a web worker, you typically only need the client library:

```
npm install @mlc-ai/web-llm
```

For CDN usage, include:

```
<script src="https://cdn.jsdelivr.net/npm/@mlc-ai/web-llm/dist/webllm.min.js"></
script>
```

The CDN build attaches a global `webllm` object which exposes the same API used in the examples below.

4. Loading and initialising a model (MLC engine)

WebLLM exposes its API through the `MLCEngine` interface. You can create an engine via the `CreateMLCEngine()` factory or by directly instantiating `MLCEngine` [5]. Model loading happens asynchronously because the engine must download the model weights and compile kernels [6].

4.1 Using the factory function

```
import { CreateMLCEngine } from "@mlc-ai/web-llm";

// Provide a callback to monitor loading progress
const initProgressCallback = (progress) => {
  console.log("Model loading progress:", progress);
};

async function initEngine() {
  const engine = await CreateMLCEngine("Llama-3.1-8B-Instruct", {
    initProgressCallback,
  });
  return engine;
}
```

The factory call returns a fully initialised engine. When calling `CreateMLCEngine`, pass the `model_id` from the `model_list` and an optional `appConfig` if you want to override WebLLM defaults (e.g., using service workers or multi-model mode).

4.2 Direct instantiation and reload

Alternatively, instantiate `MLCEngine` first and then call `reload()` to load a model [7]:

```
import { MLCEngine } from "@mlc-ai/web-llm";

const engine = new MLCEngine({ initProgressCallback });
await engine.reload("Mistral-7B-Instruct");
```

This separation is useful if you need to reuse the engine for multiple models (e.g., letting users choose from `model_list`). Call `reload()` again whenever you switch models.

5. Designing prompts to parse and analyse listings

5.1 Use system messages to set expectations

When generating chat completions, always include a **system prompt** that instructs the model to behave as a structured data extractor. For example:

```
const systemMessage = {
  role: "system",
  content: "You are a backend parser that extracts structured fields from
property listings. " +
           "For each listing, produce a JSON object containing the title,
location, price, description, and any notable features."
};
```

The system prompt is processed before user messages and influences the model's behaviour.

5.2 Provide clear examples and delimiters

Include a few examples of listings and desired outputs in your initial messages. Use delimiters such as triple backticks (` ``` `) around listings and ask the model to produce pure JSON. Example:

````
const messages = [
  systemMessage,
  { role: "user", content: "Here is a new listing:\n\n```\n3-bedroom ranch home
in Asheville NC. Listed at $480,000. Features a fenced yard and updated kitchen.
\n```\n\nPlease output a JSON object with the extracted fields." }
];
````

6. Generating responses

6.1 Standard (non-streaming) completion

Call `engine.chat.completions.create()` on the initialised engine. The model is fixed when the engine is created, so you should not pass a `model` parameter [8]. For example:

```
const response = await engine.chat.completions.create({
  messages,
  temperature: 0.2,      // lower temperature yields more deterministic outputs
  max_tokens: 400,
  response_format: { type: "json_object" }, // triggers JSON-mode if supported
});

const result = JSON.parse(response.choices[0].message.content);
console.log(result.title, result.price);
```

6.2 Streaming completion

For long listings or interactive UIs, enable streaming by passing `stream: true` [9]:

```
const chunks = await engine.chat.completions.create({
  messages,
  stream: true,
  stream_options: { include_usage: true },
});

let jsonStr = "";
for await (const chunk of chunks) {
  jsonStr += chunk.choices[0]?.delta.content || "";
  // Optionally update your UI in real time here
  if (chunk.usage) {
    console.log("Tokens used:", chunk.usage);
  }
}

const parsed = JSON.parse(jsonStr);
```

Streaming mode yields incremental tokens. Accumulate them until the end and then parse the JSON string.

7. Wiring WebLLM in a backend context

Although WebLLM runs in the browser, backend engineers often need to perform server-side validation or handle multiple concurrent listing analyses. Here are integration patterns:

1. **Worker thread or service worker** – Offload WebLLM computations from the main UI by running the engine in a Web Worker. Use `worker.postMessage()` and `onmessage` to communicate. WebLLM's documentation notes that the library supports workers for heavy computation [10].
2. **Backend-assisted caching** – The first model load can be slow because weights must be downloaded and compiled [6]. Cache the model files on a local CDN or use service workers to avoid re-downloading on subsequent sessions.
3. **Hybrid architecture** – For large analyses, you might fall back to a server-side inference service (e.g., vLLM or GPU servers) when the user's browser lacks WebGPU support. In that case, the backend can read the same prompts and call an OpenAI-compatible API on the server.

8. Custom model configuration

If the prebuilt models do not meet your requirements (e.g., you need to fine-tune on your own listing dataset), you can convert a model to MLC format and add it to WebLLM:

1. Use the MLC LLM conversion tools to export your model into the WebGPU-friendly format.
2. In your app configuration, extend the `model_list` with your new `ModelRecord` by providing the path to the weights and metadata. Place the weights on a CDN reachable by the browser.
3. Reload the engine with your custom model ID via `engine.reload("your-model-id")`.

9. Troubleshooting tips

- **Model fails to load**: Ensure the user's browser supports WebGPU. Chrome/Edge 113+ with the `--enable-features=WebGPU` flag or Firefox nightly builds support it.
- **Out-of-memory errors**: Choose a smaller or more heavily quantised model from `model_list` (e.g., 1–3 B parameters) to fit within GPU memory.
- **Invalid JSON output**: Use JSON-mode and provide explicit instructions in the system prompt. Use low temperature (0–0.3) for deterministic outputs and post-process the stream to ensure proper JSON.

10. Conclusion

WebLLM allows backend engineers to offload inference to users' browsers while still using familiar **OpenAI-style chat APIs** [11]. By inspecting the `model_list` and choosing an appropriate instruction-tuned model, you can parse and analyze property listings directly in the browser. Proper prompt engineering, streaming, and JSON-mode will help you extract structured data reliably and integrate the results into your backend workflow.

---

[1] [2] [3] [10] [11] WebLLM | Home
https://webllm.mlc.ai/