Democratizing On-Device LLM Inference with Machine Learning Compilers and Web Technologies

Charlie F. Ruan

CMU-CS-25-112

May 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Tianqi Chen, Chair
Zhihao Jia

Submitted in partial fulfillment of the requirements for the degree of Masters of Science in Computer Science.

Keywords: Large Language Models, LLM Inference, Machine Learning Compiler, WebAssembly, WebGPU, Edge Computing

To my dear friends and family.

Abstract

Large language models (LLMs) have traditionally relied on cloud-based inference due to their high computational and memory demands. However, recent advances in small LLMs and consumer hardware capabilities have made on-device inference increasingly practical. Among potential deployment targets, the web browser stands out as a uniquely compelling platform: it is universally accessible, naturally abstracts out hardware heterogeneity, requires no dependency installation for web applications, and provides a natural agentic environment for task automation.

WebLLM is a high-performance TypeScript framework that enables LLM inference entirely within client-side web browsers. WebLLM compiles LLMs ahead of time using the MLC-LLM and Apache TVM compiler stack to generate optimized WebGPU kernels and a portable WebAssembly runtime. WebLLM exposes a familiar OpenAI-style API, supports efficient GPU acceleration, and integrates seamlessly with browser environments using Web Workers and WebAssembly. To enable structured generation, which is especially challenging for small LLMs, WebLLM incorporates XGrammar, an efficient grammar-constrained decoding engine, allowing developers to enforce output formats such as JSON or DSLs with near-zero overhead. Together, these components demonstrate a path toward democratizing LLM access, making intelligent, private, and responsive AI experiences universally available through the web.

Acknowledgments

First and foremost, I would like to thank my advisor, Tianqi Chen, for inviting me into the world of machine learning systems research. His mentorship guided my career; his passion has made every project a joy; his builder's mindset has shaped my own; and his detailed feedback—on both research and engineering—has taught me how to create and steward open-source software. I could not have asked for a better mentor at this stage of my journey.

I am grateful to Zhihao Jia for his patient advice on the disaggregated serving system project. I learned a lot from his research style, especially on how to spot emerging interesting research directions.

The work presented in this thesis, WebLLM, stands on the shoulders of giants. Building MLC-LLM with the MLC community in 2023–24 was a delight; special thanks to Ruihang Lai for his immensely patient guidance and David Pissarra for the pleasant collaboration. I further thank Ruihang, Hongyi Jin, and Bohan Hou for the initial scaffolding on the WebGPU front, and every collaborator on WebLLM—Nestor Qin, Rick Zhou, and many more. I am indebted to the Apache TVM community for the infrastructure that underpins this thesis. I also thank the XGrammar team, especially Yixin Dong, for enabling reliable structured generation, and the MicroServe team—including Hongyi, Ruihang, Yingcheng Wang and Xupeng Miao—for insightful discussions.

From my MSCS program, I would like to thank David Eckhardt, Ruben Martins, and Angy Malloy for all their help.

Finally, my deepest gratitude goes to my family and close friends. I would not have come this far without their unwavering support. Thank you, Ryan, Vincent, and Danny for being such great friends. Thank you, Stephanie, for always being there for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Large language models (LLMs) have rapidly advanced in capability over the past few years, not only being used for chatbots but increasingly as components in agentic systems. Due to their massive parameter counts (e.g., 70 billion or even 405 billion), these models are typically deployed on cloud servers, which provide the computational and memory resources necessary for inference using server-grade GPUs.

Recently, however, open-source model providers have begun releasing smaller models—ranging from 7 billion to 1.5 billion parameters and even less—enabled by techniques such as model distillation ([6], [15], [18], [1]). In parallel, there has been a surge in specialized models, including those for coding and math tasks ([19], [21], [13]). On the other hand, consumer devices (e.g., Apple's M-series MacBooks) continue to grow in computing capability. These trends together make on-device LLM inference an increasingly viable and attractive direction. Potential benefits include privacy, personalization, cost reduction, and even lower latency depending on the workload. Therefore, we motivate a fundamental question:

> *How can we bring the power of modern LLMs directly onto consumer devices—without sacrificing performance and developer ergonomics?*

1.1.1 Browser as a Platform

The web browser stands out as a compelling platform for on-device LLM deployment. First, browsers provide a natural agentic environment where users already perform various tasks, such as booking calendar events or composing emails—tasks that can potentially be automated by in-browser agents. Second, browsers offer universal accessibility: users can access applications simply by opening a URL, with no need for additional installations. Finally, the browser serves as a natural abstraction layer for developers. Web technologies like WebAssembly ([7]) and WebGPU ([17]) are backend-agnostic by design, enabling LLM workloads to target a single runtime interface (e.g., WebGPU) rather than maintaining multiple backend implementations (e.g., CUDA, Metal).

1.2 Overview

In this thesis, we focus on the design and implementation of WebLLM, a high-performance in-browser LLM engine [14]. WebLLM provides a framework that empowers web developers to integrate on-device LLM inference directly into their web applications. Its design centers around three key principles: a standardized and user-friendly API, efficient GPU acceleration, and effective integration with the browser runtime environment (Chapter 3).

WebLLM builds upon machine learning compiler frameworks MLC-LLM and Apache TVM to generate optimized WebGPU kernels ([10], [3]). While MLC-LLM is best known as a backend-agnostic LLM inference engine for both cloud and edge, WebLLM repurposes it as a compiler that precompiles WebGPU kernels ahead of time (AOT) and hosts them online, ready to be downloaded by the browser when needed. TVM, besides being the compiler backend for MLC-LLM that enables its backend-agnostic compilation flow, also provides necessary runtime (host-side) support for LLM inference, such as kernel launching and tensor manipulation with TVMjs (Chapter 4).

To support structured generation—especially important for smaller LLMs that struggle with format consistency—WebLLM integrates XGrammar, an efficient structured generation framework using context-free grammars for constrained decoding ([5]). Ported to WebAssembly, XGrammar allows web applications to use LLMs to generate outputs conforming to strict grammars (e.g., JSON), expanding their use cases well beyond chatbots (Chapter 5).

Finally, we explore future directions such as hybrid inference, which disaggregates computation across cloud and edge backends to exploit the heterogeneous nature of LLM workloads ([9]). We also discuss opportunities in building agentic environments within the browser to allow LLM-based agents navigate webpages to automate tasks more easily (Chapter 6).

Chapter 2

Background

2.1 LLM Inference and On-device Deployment

LLM inference refers to the process of using a trained large language model to generate outputs (such as text completions or answers) given an input prompt. Under the hood, inference involves performing a forward pass through the model's neural network layers. Modern LLMs are typically transformer-based architectures with dozens of layers of self-attention and feed-forward networks ([16]). Performing inference on a large model is computationally intensive and requires significant memory to store the model's parameters and intermediate activations. In a cloud setting, inference is often optimized by using batches of inputs processed together on powerful GPUs, and by using techniques like caching recurrent computations across tokens to streamline the generation of long outputs.

Running LLMs locally on consumer devices (such as a laptop) introduces new challenges and opportunities. The primary challenge is the limited compute and memory resources compared to cloud servers. A high-end GPU in a server might have 80 GB of VRAM and thousands of computation units (e.g., CUDA cores), whereas a laptop GPU has much less memory and compute throughput. To make large models runnable on smaller devices, several strategies are employed:

- Model size reduction: using smaller architectures or reducing the number of parameters (for example, choosing a 7B parameter model instead of a 70B model). Techniques like model distillation is popular to enable such a reduction ([6]).
- Quantization: representing model weights with lower precision (8-bit or 4-bit integers instead of 16-bit or 32-bit floats) to shrink model memory footprint and improve throughput at the cost of some precision loss. This can dramatically reduce memory usage and can even increase speed if the hardware supports fast low-precision arithmetic.
- Efficient architectures and kernels: using optimized implementations of the model's operations (for instance, efficient attention mechanisms) that run faster on the given hardware ([4]).

These innovations have enabled surprisingly capable LLMs to run on consumer devices. Quantized models with a few billion parameters can reach interactive speeds on laptops. Local deployment offers significant benefits: privacy, since data processing stays on the user's machine;

personalization, as the model could adapt or fine-tune to user-specific data or preferences that never leave the device; and availability, since the model can function without internet access or when cloud services are offline. It also reduces reliance on a central service and can save cost for users (especially if the alternative is paying for API calls to a cloud LLM). However, local inference must work within the constraints of device hardware, which may involve trade-offs in model size or speed, and typically must handle one request at a time.

2.2 Web Technologies

2.2.1 WebAssembly

WebAssembly (WASM) is a binary instruction format for a virtual machine that runs in web browsers [7]. It is designed as a portable target for compiling high-level languages like C, C++ or Rust, so that they can execute on the web at near-native speed. WebAssembly is safe (running in a sandboxed environment), compact, and efficient. In the context of machine learning, heavy computations that would be too slow in JavaScript can be implemented in C++ or Rust, compiled to WASM, and then run in the browser faster than pure JavaScript would allow. In WebLLM, many parts of the runtime (especially those not run on the GPU) are handled by a WASM runtime library TVMjs, which is essentially a version of the TVM runtime for JavaScript environments [3]. This library can perform tensor allocations and copies, KV cache management, and kernel invocation, all within the browser environment. Because WebAssembly is platform-neutral, the same WASM code runs on Windows, macOS, Linux, or mobile browsers, ensuring broad compatibility.

2.2.2 WebGPU

WebGPU is a new Web standard (and the successor to WebGL) that provides a modern, high-performance API for GPU programming in web browsers [17]. Whereas WebGL was primarily aimed at drawing graphics, WebGPU is designed to expose general-purpose GPU computing capabilities as well as advanced graphics. WebGPU's API is based on contemporary GPU APIs like Vulkan, Metal, and Direct3D 12, and it allows developers to write compute shaders—programs that run on the GPU for arbitrary computations (not just graphics). This is crucial for machine learning workloads, as we can offload tensor operations to the GPU for acceleration. With WebGPU, a web application can request the creation of a GPU device, allocate GPU buffers, and run compiled shader code. In practice, one writes shader programs in a language called WGSL (WebGPU Shading Language), which is then compiled by the browser's graphics driver to the native GPU instructions. WebLLM uses WebGPU to run the neural network operations (matrix multiplications, attention, etc.) on the graphics hardware, which is orders-of-magnitude faster than doing the same operations on the CPU in JavaScript.

One of the powerful aspects of WebGPU in the context of deploying ML models is its backend-agnostic nature. A WebGPU shader can run on any GPU (from any vendor) that the browser supports, without the developer needing to tailor the code to each specific GPU type. This means a single implementation can target many platforms. However, a challenge is that, un-

like established ML frameworks on native platforms (which have libraries like cuBLAS, cuDNN, etc. for NVIDIA GPUs or similar libraries for other backends), the web environment doesn't come with an existing library of highly optimized tensor operations for WebGPU. This is where ML compilers and frameworks step in—they must generate efficient GPU code because we cannot rely on linking against vendor-specific libraries in the browser. WebLLM's use of a compiler to produce WebGPU code aims to fill this gap, by compiling WebGPU kernels that are potentially as optimized as if one had hand-tuned them.

2.2.3 Web Worker

The W3C specification defines Web Workers as an API for spawning background worker scripts that run alongside the main page script. In essence, a web worker executes code in a separate thread, distinct from the single-threaded JavaScript main thread, which handles the UI. This mechanism allows computationally heavy tasks to be offloaded to a background thread, preventing the user interface from freezing. Communication between the main script and a web worker occurs via asynchronous message passing. Web workers effectively introduce a form of multi-threaded execution in browsers, which is especially useful for performance-intensive operations. Developers use them to handle tasks like data parsing, image processing, or complex computations without interrupting user interactions. WebLLM leverages web workers to run LLM inference in a background thread, avoiding blocking the UI due to the heavy LLM computations.

2.3 Machine Learning Compilers

Machine learning compilers are specialized compilers or frameworks that take high-level descriptions of ML models (often as computational graphs) and generate low-level optimized code for a given hardware target. Examples of ML compilers include torch.compile and Apache TVM ([2], [3]). The motivation for ML compilers is that modern deep learning models consist of many operations (matrix multiplications, convolutions, element-wise ops, etc.), and there are numerous possible ways to implement and optimize these operations on different hardware. Rather than writing each kernel by hand for each platform, a compiler can automate this process: it can apply optimizations like kernel fusion (merging multiple operations into one to avoid intermediate memory writes), tiling and vectorization (to better use caches or SIMD units), unrolling, memory layout transformations, and use hardware-specific intrinsics. By doing so, the compiler can often approach or exceed the performance of hand-tuned libraries, especially for novel model architectures where hand-written kernels do not yet exist.

In WebLLM, the use of a compiler is essential because we do not have access to highly optimized BLAS libraries or vendor-specific ML runtimes in the browser. By using TVM and MLC-LLM, WebLLM essentially pre-builds its own optimized kernels. The TVMjs runtime then loads those kernels in the browser and executes them. This approach was shown to yield performance close to native: WebLLM retained around 71–80% of the throughput of a native engine on the same device, which is impressive given the overheads of running inside a browser.

Chapter 3

WebLLM Core Design

3.1 Challenges and Design Principles

Building a high-performance LLM inference engine that runs entirely inside a web browser required us to address several key challenges. We identified three overarching challenges and corresponding design principles in the development of WebLLM:

1. **Standardized, Easy-to-use API** Web developers should be able to use the on-device LLM as seamlessly as they would call a cloud API. This means exposing a clear and simple interface for sending prompts to the model and receiving generated text, ideally following conventions that developers are already familiar with. The design principle here is to abstract away the complexity of the underlying model and compiler, and present a friendly API (we chose to emulate OpenAI's API style) that returns results in a straightforward format (in JSON). By standardizing the API, we also future-proof the system: improvements or changes under the hood shouldn't require changes in how developers integrate the engine.
2. **Adaptation to the Browser Runtime** The browser environment is quite different from a typical native runtime. There are restrictions on threading, memory, and execution time (to avoid freezing the page). Also, computations run in the main thread can block the user interface, harming the user experience. Thus, WebLLM's design principle is to fully integrate with the browser runtime model, using features like Web Workers (background threads) to offload heavy computations away from the main UI thread. In addition, WebLLM extensively leverages WebAssembly to port existing codebase implemented in C++ for both code reuse and efficient execution of non-GPU workload in LLM inference.
3. **Efficient GPU Acceleration** Performance is paramount given the scale of LLMs. To get usable speeds, we must efficiently use the device's GPU whenever available. However, the challenge is that in a browser we cannot simply use existing GPU libraries—we must rely on WebGPU and our own optimized kernels. The design principle is to embrace ML compilers to generate high-performance GPU code. We also aimed to achieve near-native performance compared to that of a native backend on the same device.

These principles guided the architecture of WebLLM. By focusing on API simplicity, environment integration, and performance, we set the stage for an engine that could be both

developer-friendly and efficient in practice.

3.2 Systems Architecture

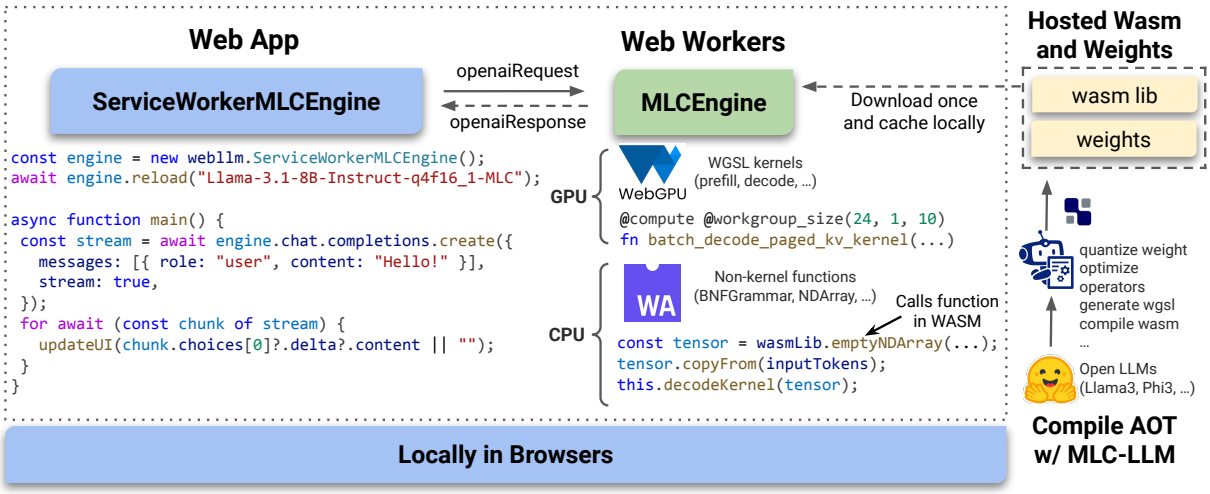


Figure 3.1: Systems Architecture of WebLLM [14].

To address the challenges above, WebLLM's system architecture (illustrated in Figure 3.1) cleanly separates responsibilities into three main components:

1. **`ServiceWorkerMLCEngine` (User-Facing Engine)** This is a lightweight engine instance that lives in the web application's context. The web developer interacts with this engine as if it were a remote service. From the developer's perspective, they initialize the engine, ask it to load a particular model (which triggers downloading the model artifacts or loading from browser cache), then send inference requests and receive responses. The `ServiceWorkerMLCEngine` is designed to feel like calling an endpoint: the API calls and responses use a consistent JSON schema that mirrors OpenAI's API. Internally, however, this engine does not do heavy work; it primarily forwards requests to the background worker and streams back results.
2. **`MLCEngine` in Web Worker (Backend Engine)** This is the component that actually carries out the inference workload. It runs in a Web Worker thread, which means it operates off the main thread, ensuring that intensive computation does not block the UI. When the `ServiceWorkerMLCEngine` receives a request, it message-passes it to the `MLCEngine` in the worker. The `MLCEngine` manages the loaded model, runs the inference by invoking the appropriate compiled kernels (via the TVM runtime), and streams the output tokens back to the frontend engine as they are produced. By encapsulating the `MLCEngine` in a worker, we isolate the computational workload.
3. **Ahead-of-Time Compiled Kernels and Artifacts** Such artifacts include the precompiled WebGPU kernels, compiled WebAssembly for runtime support, and LLM weights converted to MLC-LLM's format. All of these artifacts are compiled ahead of time by MLC-

LLM and TVM. They are hosted online (GitHub and HuggingFace) and downloaded once an engine requests to load the model. They are cached in the end user's browser after the initial download. Therefore, the first time an application uses a model, there is a loading delay, but afterwards it can run offline. Besides, developers can easily host new models without needing to ship huge binaries in the app bundle; they just point WebLLM to the URL of the model artifacts.

This architecture is illustrated in Figure 3.1. In summary, the web application talks to a local service endpoint, which delegates to a background thread running the model, which in turn uses precompiled GPU code to do the heavy workload. This design maps to the three challenges: the Service Worker interface provides the standardized API; the use of a Web Worker adapts to the browser runtime model; and the precompiled kernels enable efficient GPU acceleration.

3.3 API Design

WebLLM's API is modeled after the well-known OpenAI API as shown below. The motivation behind this design choice is to lower the barrier for adoption: many developers are already familiar with OpenAI API and how the requests and responses look like (with knobs such as `n`, `temperature`, etc.). By providing a similar interface locally, WebLLM allows developers to swap out the backend from a cloud API to an on-device engine with minimal changes to their code. Besides, the use of an engine makes the API behave like an endpoint (Figure 3.2). The JSON-in-JSON-out interface allows the engine behaviors to be easily well-defined.

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

const engine = new MLCEngine();
async function main() {
 await engine.reload("Llama-3-8B-Instruct-MLC");
 const stream = await engine.chat.completions.create({
   messages: [{ role: "user", content: "Introduce yourself!" }],
   stream: true, temperature: 1.2,
 });
 let finalMessage = "";
 for await (const chunk of stream) {
   finalMessage += chunk.choices[0]?.delta?.content || "";
 }
}
```

Figure 3.2: WebLLM engine's API follows that of OpenAI. A streaming request implemented in TypeScript.

Such an API design can also easily extend to new features. For instance, WebLLM supports structured generation that enforces the LLM to generate response in a specified format (e.g., JSON). With a single line of change, the code in Figure 3.3 ensures that the output is in JSON format.

The same API can also extend to other features such as supporting embedding models, vision-language models, loading multiple models in the same engine (e.g., for RAG), and function calling.

```
const stream = await engine.chat.completions.create({
   messages: [{role: "user", content: "Introduce yourself!"}],
   stream: true, temperature: 1.2,
   response_format: { type: "json_object" },
});
```

Figure 3.3: The same request but ensures the output to be in JSON.

3.4 GPU Acceleration

Achieving good performance for LLM inference in the browser hinges on making maximal use of the GPU via WebGPU. As shown in Figure 3.1, WebLLM leverages MLC-LLM and Apache TVM to achieve this. WebLLM treats the MLC-LLM framework as a black box, which compiles any LLM it supports into kernels for a specified backend (e.g., CUDA, ROCm, Metal), and WebGPU is one of the backends supported by MLC-LLM. The user (or WebLLM developer) feeds a model from HuggingFace to the MLC-LLM framework, which produces a set of optimized WebGPU kernels for the model, CPU runtime support compiled into WebAssembly, and model weights converted into MLC-LLM's format. We elaborate on this in Chapter 4.

3.5 Adapting to Browser Runtime

While most LLM frameworks are implemented in either Python or C++, WebLLM is implemented in TypeScript and runs in the browser environment. This unconventional environment requires us to adapt to the unique characteristics of browsers. We leverage three key web technologies in WebLLM.

WebGPU We have discussed extensively about WebGPU so far. It is a required technology because we want to leverage GPUs for the heavy LLM computations whenever possible. Besides performance, WebGPU provides a natural abstraction for the wide range of consumer devices. This standard allows us to only compile a single set of kernels that can be deployed on all devices, regardless of their native backends.

WebAseembly While WebGPU kernels handle the heavy lifting of the LLM workloads, WebLLM also relies on WebAssembly for various aspects of the system. This is because there are many non-GPU workloads required in LLM inference. TVMjs is a web runtime library compiled from the C++ code in TVM into WebAssembly. It provides a set of functions for allocating tensors, launching GPU kernels, and performing any CPU-side computation needed. In addition, for structured generation, we compile the XGrammar library into WebAssembly as well, for performant constrained decoding support. This will be elaborated in Chapter 5.

Web Worker Finally, as discussed in Section 3.2, web applications prefer offloading the heavy computation to an asynchronous thread called the web worker. The goal is to prevent the heavy workloads from blocking the UI flow, which may cause the web page to freeze. The majority

of WebLLM's workload is carried out in the web worker, as we have a frontend engine and a backend engine, as shown in Figure 3.1.

3.6 Evaluation of WebLLM

We evaluate the performance of WebLLM. This is especially interesting because, while WebGPU provides a convenient standard that can work across different devices, such abstraction must come at a cost. That is, we cannot use backend-specific features, nor can we tune to the preference of a specific backend.

Model	**WebLLM** (tok/s)	**MLC-LLM** (tok/s)	**Perf. Retained**
Llama-3.1-8B	41.1	57.7	71.2%
Phi-3.5-mini (3.8B)	71.1	89.3	79.6%

Table 3.1: Performance comparison with decoding speed between WebLLM (v0.2.75) and MLC-LLM (at commit d23d6f5). WebLLM runs on Chrome Canary 133.0.6870.0 (arm64). Adopted from [14].

We run WebLLM and MLC-LLM on the same MacBook Pro M3 Max 64GB device. While WebLLM uses WebGPU kernels, MLC-LLM uses the native Metal kernels. Both kernels are generated by MLC-LLM. The result demonstrates that WebLLM can maintain up to 80% of the native performance. There are still many potential techniques to further close this gap, including offloading more workload to the GPU (e.g., sampling and penalties application) and leveraging more advanced WebGPU features (e.g., subgroup shuffle). We could also compile two sets of WebGPU kernels: one for the lower-end consumer devices like phones, and one for the more performant devices like MacBooks, where the latter allows more aggressive WebGPU settings such as a higher number of threads per thread block.

Chapter 4

Machine Learning Compiler and Runtime

4.1 MLC-LLM as a Compiler for WebLLM

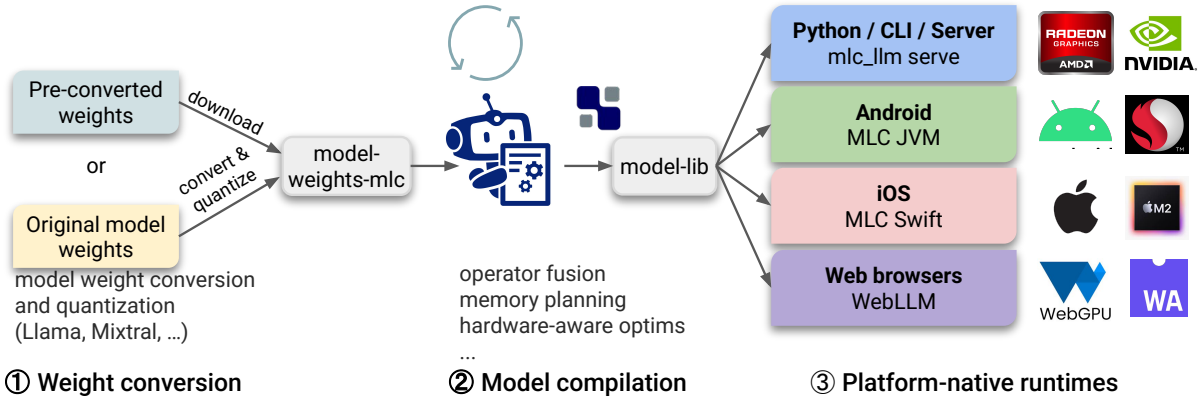


Figure 4.1: Compilation flow of MLC-LLM. Adapted from [10].

MLC-LLM is a framework that is developed to enable LLM inference on a variety of platforms (from cloud GPUs to mobile phones) using a unified approach based on ML compilation. It stands for "Machine Learning Compilation for LLMs." At its core, MLC-LLM takes a model implemented in a domain-specific language (DSL) in Python and compiles that model to a chosen target backend via TVM (Figure 4.1). In the case of WebLLM, we use MLC-LLM to target the WebGPU backend. Below we demonstrate an example model definition in MLC-LLM (the input, Figure 4.2), and the corresponding WebGPU kernel generated (the output, Figure 4.3).

While MLC-LLM is an LLM serving engine itself, WebLLM views it more as a compiler for WebGPU kernels. However, we could also potentially port MLC-LLM's runtime logics (e.g., serving engine state management, request scheduling) to WebAssembly to use in WebLLM. WebLLM did not go with that option since most on-device use cases do not require sophisticated request batching logics (unlike the cloud counterpart). Therefore, WebLLM implemented the engine logic in a lightweight format in TypeScript instead of relying on MLC-LLM.

```
def batch_decode(self, input_embeds, paged_kv_cache, logit_pos):
  op_ext.configure()
  hidden_states = self.model(input_embeds, paged_kv_cache)
  if logit_pos is not None:
    hidden_states = op.take(hidden_states, logit_pos, axis=1)
  logits = self.get_logits(hidden_states)
  return logits
```

Figure 4.2: Example implementation of a decoding function with DSL in MLC-LLM.

```
fn batch_decode_paged_kv_kernel(
  @builtin(workgroup_id) blockIdx : vec3<u32>,
  @builtin(num_workgroups) gridDim : vec3<u32>,
  @builtin(local_invocation_id) threadIx : vec3<u32>,
) {
  if (blockIdx.z * gridDim.x + blockId.x > podArgs.packGridDimX) {
    return;
  }
  let v__1 : i32 = i32(blockIdx.z * gridDim.x + blockIdx.x);
  var kv_chunk_len : array<i32, 1>;
  var st_m : array<f32, 1>;
  var st_d : array<f32, 1>;
  var O_local : array<vec4<f32>, 1>;
  ...
}
```

Figure 4.3: The corresponding WebGPU kernel generated by MLC-LLM's compilation flow.

4.2 TVMjs

Besides WebGPU kernels (generated by MLC-LLM) and engine serving logic (re-implemented in WebLLM), we also need other runtime components. Specifically, the host needs to invoke the generated WebGPU kernels during runtime. This also requires manipulating tensors (which are passed as inputs and outputs of kernels) and copying from JavaScript arrays to such tensors (hence host-device communication).

Fortunately, TVM already provides a set of viable runtime supports, where the `PackedFunc` mechanism plays a crucial role. Since TVM's runtime logics are all implemented in C++, we can port them to WebAssembly with Emscripten [20]. In addition, we also need WASI to call into system library calls (`malloc`, `stderr`) [8]. Packing these components, along with other support such as WebGPU device management, we have the TVMjs library.

Below is an example of how WebLLM leverages TVMjs to call the compiled kernels (Figure 4.4). `wasmSource` is essentially the artifact compiled by MLC-LLM in Figure 4.1. Functions like `this.tvm.empty()` call the underlying C++ implementation in TVM runtime, while `this.embed()` points to the WebGPU kernels.

```
this.tvm = await tvmjs.instantiate(new Uint8Array(wasmSource));
this.device = this.tvm.webgpu();

/**
 * Given input tokens, return embeddings of them
 * by calling the embed kernel.
 */
private getTokensEmbeddings(
  inputTokens: number[]
): tvmjs.NDArray {
  this.tvm.beginScope();
  const inputData = this.tvm.empty(
    [inputTokens.length],
    "int32",
    this.device,
  );
  inputData.copyFrom(inputTokens);
  const embed: tvmjs.NDArray = this.tvm.detachFromCurrentScope(
    this.embed!(inputData, this.params),
  );
  this.tvm.endScope();
  this.tvm.attachToCurrentScope(embed);
  return embed;
}
```

Figure 4.4: An example of how WebLLM uses TVMjs for host-side runtime support.

Chapter 5

Structured Generation with XGrammar

5.1 Motivation and Background

Modern LLMs not only emit free–form natural language alone, they are increasingly expected to return *structured* responses such as JSON objects, SQL queries, or domain-specific language that can be consumed programmatically. Smaller on-device models—the focus of this thesis—tend to break syntax more often than their larger cloud counterparts, making it difficult to deploy LLMs as a standardized tool (rather than just a chatbot) in consumer devices. Therefore, we need a mechanism to enforce structured generation.

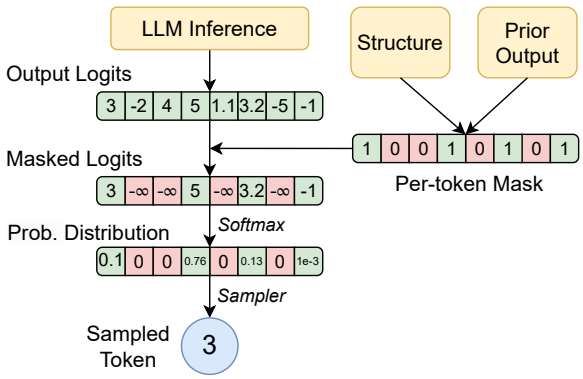


Figure 5.1: Constrained decoding. Adapted from [5].

The standard technique is **constrained decoding**: before sampling each token, the engine masks out vocabulary items that would violate a user-specified grammar (Figure 5.1). For expressive formats, we need the full power of a **context-free grammar (CFG)**, which can be executed via a byte-level **pushdown automaton (PDA)** (Figures 5.2, 5.3). During an LLM generation, each autoregressive step will correspond to a state in the automata. Then, at each step, we check each token in an LLM's vocabulary. If the token would lead to an invalid sequence, it will be masked before sampling. Then we perform sampling, and the sampled token (guaranteed to obey the specified structure) will advance the state of the automata, repeating the steps until the generation stops.

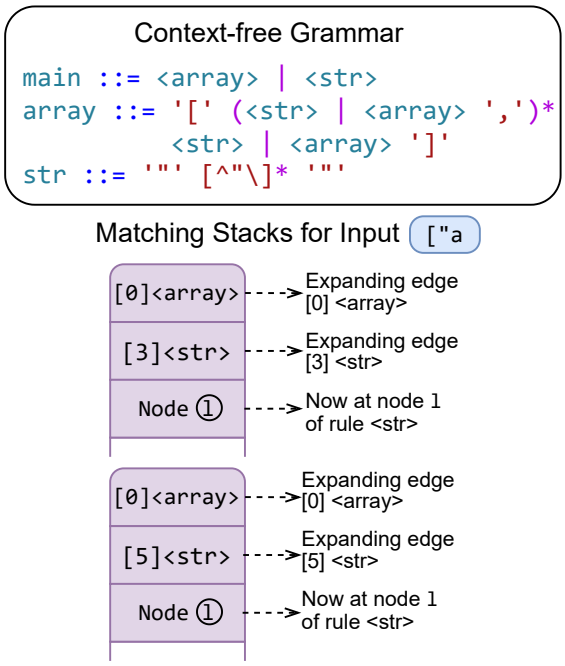


Figure 5.2: A context-free grammar with an example matching stack. The CFG is translated to the pushdown automaton in Figure 5.3. Adapted from [5].

However, this checking can be very slow if done naively, especially because modern LLMs have a large vocabulary size (e.g., 128K for Llama3.2). XGrammar is built to support structured generation with LLM efficiently.

5.2 XGrammar Core Design

XGrammar is a portable C++ library designed to enable LLM structured generation with near-zero overhead. It leverages a combination of techniques such as pre-computation, data-structure engineering, and CPU-GPU overlapping. The key flow of XGrammar is illustrated in Figure 5.3. Below, we present the key insights and optimizations from XGrammar.

Adaptive Token-Mask Cache The key observation is that, at each PDA node, usually more than 99% of the tokens can be classified as accept/reject without inspecting the stack. XGrammar pre-computes these *context-independent tokens* once, stores them in a compressed cache, and only checks the *context-dependent* remainder at runtime, dramatically decreasing the overhead.

Context Expansion For each rule, XGrammar statically infers the set of suffixes that can legally follow it. Any token whose suffix cannot match is rejected up-front, cutting context-dependent tokens by up to 90%.

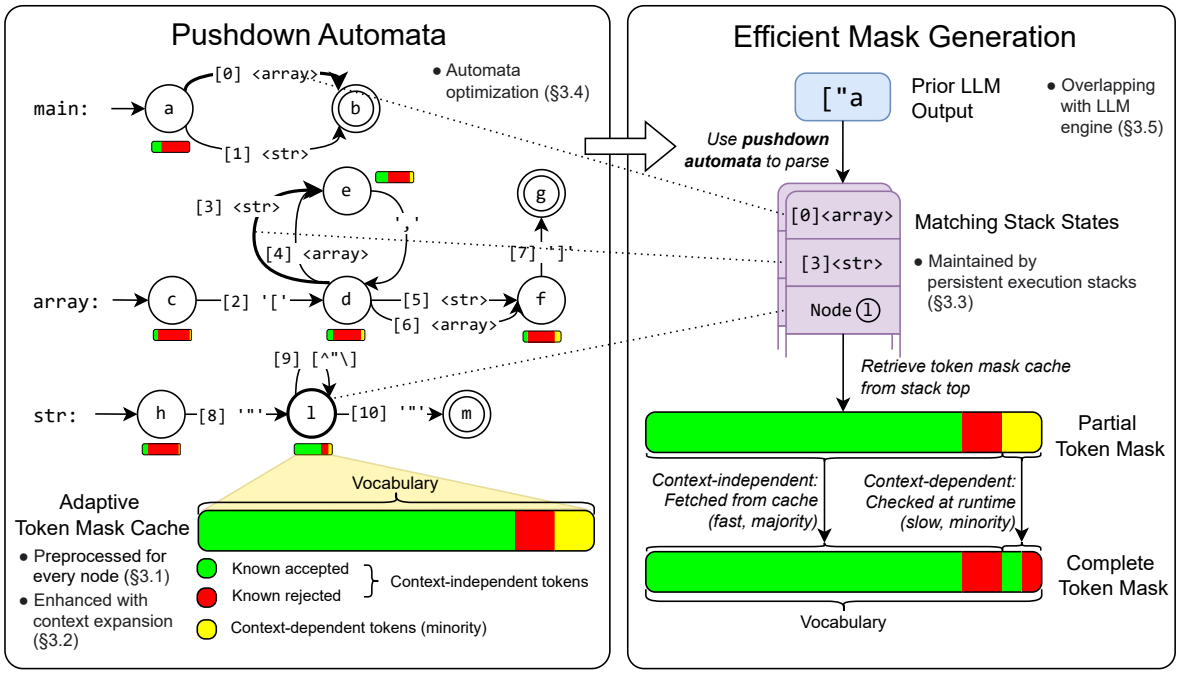


Figure 5.3: Flow of XGrammar. Adapted from [5].

Persistent Execution Stack Runtime stacks (including parallel stacks arising from ambiguous grammars) are stored in a tree that supports *fork* and *rollback*. This de-duplicates common prefixes when scanning many candidate tokens, reducing memory footprint and enabling advanced features such as speculative decoding.

Automata-Level Optimizations Inlining tiny fragment rules and merging equivalent PDA states shrink both nodes and edges, reducing stack explosion and token checks further.

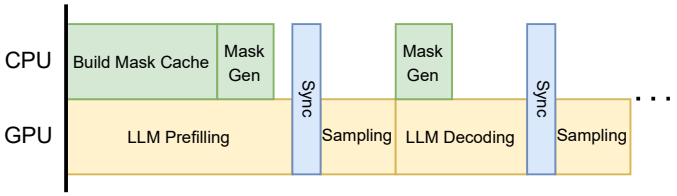


Figure 5.4: Overlapping of LLM GPU workload and XGrammar CPU workload. Adapted from [5].

CPU-GPU Overlap The token checking and automata maintenance are purely done on the CPU. Therefore, XGrammar overlaps mask generation with GPU compute and only synchronizes before sampling. Because a mask is ready *before* GPU decoding finishes, structured generation adds virtually no critical-path time (Figure 5.4).

5.3 Evaluation of XGrammar on WebLLM

Since XGrammar is implemented purely in C++, we leverage Emscripten [20] to port it to WebAssembly and build a TypeScript library around it that can be used by WebLLM.

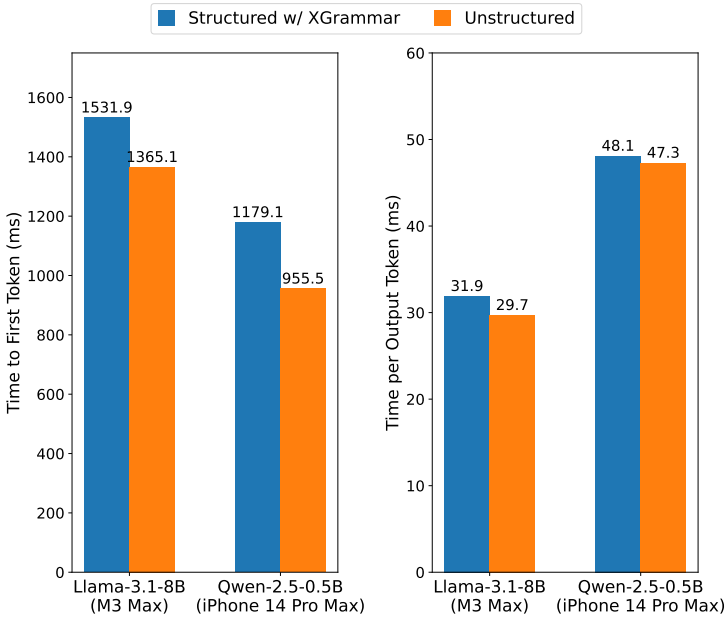


Figure 5.5: Performance on WebLLM with structured generation turned on and off. Adapted from [5].

We evaluate the performance of WebLLM when using XGrammar and without using XGrammar on both MacBook M3 Max and iPhone 14 Pro Max. Across an 8B model and a 0.5B model, we demonstrate that the overhead of XGrammar is near-negligible (especially for the time per output token).

Chapter 6

Future Directions

The combination of WebLLM, MLC-LLM, Apache TVM, and XGrammar demonstrates that high-quality language-model inference is now possible directly inside consumer browsers. Nonetheless, several open questions remain to be worth exploring. This chapter outlines two particularly interesting avenues for future work:

- **Hybrid inference**, which dynamically partitions computation between the client and the cloud.
- **In-browser agent environments**, specifically providing small local models with semantically rich coarse-grained actions for web tasks automation.

6.1 Hybrid Inference

While WebLLM focuses on fully on-device execution, many real-world applications can benefit from an edge–cloud strategy that adaptively offloads portions of the workload. Recent work such as Minions [12] illustrates how a carefully designed hybrid architecture can improve both cost efficiency—by invoking cloud models only when necessary—and *privacy*—by keeping sensitive user data local.

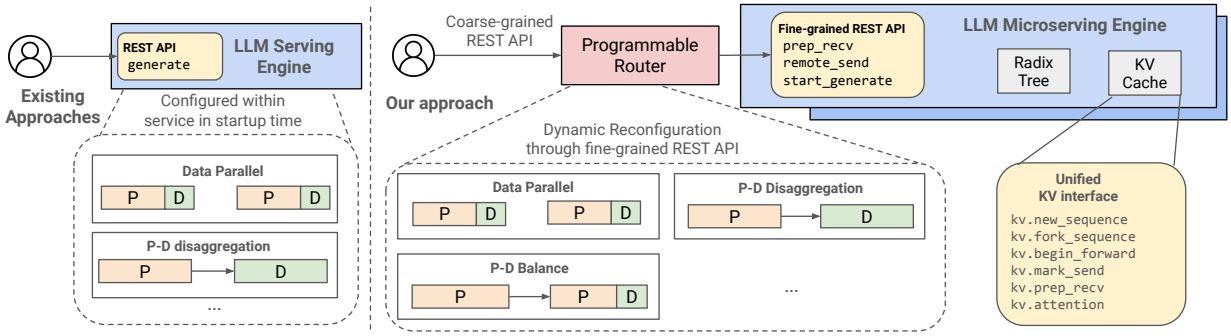


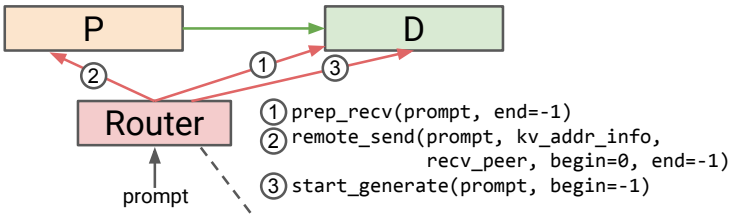
Figure 6.1: Systems architecture of MicroServe. Adapted from [9].

MicroServe is a system that exemplifies this disaggregated inference paradigm by introducing a microservice-inspired architecture for LLM inference. Rather than treating the model as

a monolithic black box, it decomposes inference into fine-grained microservices that cooperate over a lightweight REST interface (Figure 6.1). Concretely, the framework exposes three primitive APIs:

- `prep_recv`: prepare to receive key/value cache (essentially allocate and get ready to accept some portion of the prompt's KV).
- `remote_send`: prefill the prompt and send a portion of (or all of) the key/value cache from one engine to another.
- `star_generate`: begin the generation (which could include both finishing prefilling the prompt and then decoding tokens).

Compared with a coarse-grained request/response interface, these primitives give system builders explicit control over how work is partitioned. They enable, for example, *prefill–decode disaggregation*, distributed KV-cache management, or elastic data-parallel scaling—each of which can be expressed by different compositions of the three calls. Figure 6.2 demonstrates achieving prefill-decode disaggregation using MicroServe's APIs.




```
async def prefill_decode(
  request
) -> AsyncGenerator[Chunk, Any]:
  kv_addr_info, _ = decode_engine.prep_recv(
    request.prompt, end=-1, seq_id=request.id
  )
  await prefill_engine.remote_send(
    request.prompt, kv_addr_info, decode_engine_id,
    begin=0, end=-1, seq_id=request.id
  )
  async for chunk in decode_engine.start_generate(
    request.prompt, begin=-1, seq_id=request.id
  ):
    yield chunk
```

Figure 6.2: Prefill-decode disaggregation expressed in MicroServe's APIs. Adapted from [9].

Extending WebLLM with a MicroServe-style runtime would let browsers collaborate with remote engines opportunistically: a local model could handle typical requests offline or under privacy constraints, while seamlessly delegating complex planning or long-context processing to more capable cloud models. Achieving this will require a scheduling policy that balances latency, bandwidth, output quality, and cost.

6.2 Browser Agent Environment

Beyond hybrid inference, another path toward truly agentic web applications is to enrich the action space available to in-browser LLMs. Existing work such as Browser Use [11] lets a model operate directly on low-level actions such as "`click` on a certain element". Although expressive, this granularity places a heavy reasoning burden on small on-device models, which must plan long sequences of atomic manipulations.

We hypothesize that introducing a library of **coarse-grained, semantically meaningful actions** can bridge this capability gap. For a given website, developers—or even the broader community—could implement JavaScript "macros" such as `open-menu`, `type-search-query`, or `add-item-to-cart`. Exposing these macros as callable functions reduces multi-step planning to a single action selection, allowing lightweight models to succeed where they would otherwise struggle.

In the longer term, a powerful foundation model could automatically explore a site and synthesize the macro library. Small client-side models would then invoke these macros through function calling (validated by XGrammar). Key research questions include automated discovery of robust action abstractions, security of generated JavaScript, and adaptive fallback strategies when a macro fails due to page layout changes.

Chapter 7

Conclusion

This thesis examined whether modern large language models (LLMs) can run entirely inside consumer devices' web browsers without prohibitive loss of performance or developer ergonomics. The question arises from two converging trends: (i) smaller LLMs now deliver strong quality, and (ii) laptops and phones continue to gain compute power. Because browsers are universally available and backend-agnostic, they present a natural platform for on-device inference.

We provide a positive answer with WebLLM, an open-source TypeScript system that achieves near-native performance (up to 80%) for on-device inference by leveraging machine learning compilers (MLC-LLM and Apache TVM) and web technologies (WebGPU and WebAssembly). To broaden practical use cases, we integrated XGrammar for grammar-constrained decoding, enabling reliable structured outputs such as JSON.

Taken together, these contributions lower technical barriers to privacy-preserving, cost-effective, and universally accessible AI applications on everyday devices. We hope this work encourages further exploration of on-device inference and its role in democratizing LLMs and AI.

Bibliography

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024. 1.1

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024. 2.3

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018. 1.2, 2.2.1, 2.3

[4] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022. 2.1

[5] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models, 2024. URL `https://arxiv.org/abs/2411.15100`. (document), 1.2, 5.1, 5.2, 5.3, 5.4, 5.5

[6] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. 1.1, 2.1

[7] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 185–200, 2017. 1.1.1, 2.2.1

[8] Pat Hickey, Jakub Konka, Dan Gohman, Sam Clegg, Andrew Brown, Alex Crichton, Lin Clark, Colin Ihrig, Peter Huene, Yuji YAMAMOTO, Denis Vasilik, Josh Triplett, Sergey Rubanov, Syrus Akbary, Mike Frysinger, Aaron Turner, Alon Zakai, Andrew Mackenzie, Benjamin Brittain, Casper Beyer, David McKay, Leon Wang, Marcin Mielniczuk, Mendy Berger, PTrottier, Piotr Sikora, Till Schneidereit, Katelyn Martin, and Nasso. WebAssembly/WASI: snapshot-01. URL `https://doi.org/10.5281/zenodo.4323447`.

4.2

[9] Hongyi Jin, Ruihang Lai, Charlie F. Ruan, Yingcheng Wang, Todd C. Mowry, Xupeng Miao, Zhihao Jia, and Tianqi Chen. A system for microserving of llms, 2024. URL https://arxiv.org/abs/2412.12488. (document), 1.2, 6.1, 6.2

[10] MLC team. MLC-LLM, 2023-2025. URL https://github.com/mlc-ai/mlc-llm. (document), 1.2, 4.1

[11] Magnus Müller and Gregor Žunič. Browser use: Enable ai to control your browser, 2024. URL https://github.com/browser-use/browser-use. 6.2

[12] Avanika Narayan, Dan Biderman, Sabri Eyuboglu, Avner May, Scott Linderman, James Zou, and Christopher Re. Minions: Cost-efficient collaboration between on-device and cloud language models, 2025. URL https://arxiv.org/abs/2502.15964. 6.1

[13] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. 1.1

[14] Charlie F. Ruan, Yucheng Qin, Xun Zhou, Ruihang Lai, Hongyi Jin, Yixin Dong, Bohan Hou, Meng-Shiun Yu, Yiyan Zhai, Sudeep Agarwal, Hangrui Cao, Siyuan Feng, and Tianqi Chen. Webllm: A high-performance in-browser llm inference engine, 2024. URL https://arxiv.org/abs/2412.15803. (document), 1.2, 3.1, 3.1

[15] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025. 1.1

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 2.1

[17] World Wide Web Consortium (W3C). Webgpu. https://www.w3.org/TR/webgpu/, 2023. Accessed: 2025-05-06. 1.1.1, 2.2.2

[18] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024. 1.1

[19] An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024. 1.1

[20] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312, 2011. 4.2, 5.3

[21] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024. 1.1