

An Architecture for Web-Based Distributed LLM Inference

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Gabriel Kitzberger, BSc

Matrikelnummer 12024014

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar

Mitwirkung: Dr. Alireza Furutanpey

Wien, 13. Februar 2026

Gabriel Kitzberger

Schahram Dustdar

An Architecture for Web-Based Distributed LLM Inference

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Gabriel Kitzberger, BSc

Registration Number 12024014

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar

Assistance: Dr. Alireza Furutanpey

Vienna, February 13, 2026

Gabriel Kitzberger

Schahram Dustdar

Declaration of Authorship

Gabriel Kitzberger, BSc

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work – including tables, maps and figures – which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix “Overview of Generative AI Tools Used” I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, February 13, 2026

Gabriel Kitzberger



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First, I would like to thank Alireza Furutanpey for supporting me throughout this thesis. Your insights into the scientific process have been invaluable; you always pushed me in the right direction while still letting me come to the correct conclusions myself. I can remember how little I knew about scientific work before starting this thesis. Thank you, I found in you a person that I can strive to be.

I also thank Pantelis Frangoudis for providing feedback during the initial stages and Alexander Knoll for supporting me in setting up the test bed for my evaluation.

I am grateful to Schahram Dostar for providing the framework in which this thesis was built. The DSG team is a wonderful collection of people with the right amount of resources to produce great results. The atmosphere in the team was fantastic and I believe the positive culture is a reflection of your leadership.

Lastly, I am deeply grateful to my family and friends. Thank you for always being there and listening when stress got high or something did not go as planned.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Large Language Models (LLMs) umfassen viele Milliarden Parameter und benötigen für Inferenz Hunderte Gigabyte an Arbeitsspeicher. Dies beschränkt den Zugang zu den modernsten Modellen auf große Organisationen mit spezialisierter Server-Hardware. Verteilte LLM-Inferenz ermöglicht hingegen, Modelle über verschiedene Geräte hinweg bereitzustellen, die für ein einzelnes System zu groß wären. Bestehende Ansätze sind jedoch oft limitiert: vLLM setzt homogene Hardware voraus, während Frameworks wie Petals eine aufwendige manuelle Konfiguration erfordern.

In dieser Arbeit wird eine Architektur für webbasierte, verteilte LLM-Inferenz vorgestellt. Durch die Nutzung des Browsers als Plattform ermöglichen wir Zero-Cost-Deployment: Teilnehmer müssen lediglich eine Webseite aufrufen, woraufhin ein Rust-basierter Orchestrator automatisch Modellpartitionen zuweist. Das System nutzt ONNX Runtime Web für browserbasierte Inferenz, WebGPU zur Hardwarebeschleunigung sowie Protocol Buffers über WebSockets für eine effiziente Kommunikation. Um der Heterogenität der Hardware gerecht zu werden, wurde ein dynamischer Partitionierungsalgorithmus implementiert, der End-to-End-Latenz minimiert.

Unsere Experimente zeigen, dass der Orchestrierungsserver mit weniger als 0,15% der Inferenzzeit einen vernachlässigbaren Overhead verursacht. In Umgebungen mit hoher Bandbreite bleibt der Netzwerkanteil bei bis zu 10 Teilnehmern unter 25%, was belegt, dass das System primär rechengebunden arbeitet. Die Integration der Token-Dekodierung direkt in den ONNX-Graphen reduziert den Datentransfer um 75 MB pro Anfrage, während Prefix-Caching Time to First Token (TTFT) um bis zu 30% verbessert. Obwohl der Durchsatz unter dem lokaler Inferenz-Engines wie Ollama liegt, zeigen wir, dass webbasierte verteilte Inferenz eine praktikable, konfigurationsfreie Lösung für den Zugang zu großskaligen Modellen darstellt.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

LLMs reach many billions of parameters, requiring hundreds of gigabytes of memory to run inference for the largest frontier models. This restricts access to large organizations using server-grade hardware. Distributed LLM inference enables multiple devices to serve models that would be too large for just one device. However, existing solutions remain restrictive. In particular, common inference servers, such as vLLM, assume homogeneous hardware, whereas frameworks like Petals necessitate manual configuration and setup.

To this end, we propose an architecture for web-based distributed LLM inference. Using the browser as a distribution tool, we enable zero-cost deployment; participants connect to a website, and a Rust-based orchestrator automatically assigns model partitions. The system uses ONNX Runtime Web for browser-based inference, WebGPU for acceleration, and Protocol Buffers over WebSockets for efficient communication. To accommodate hardware heterogeneity, we implement a dynamic partitioning algorithm that minimizes end-to-end latency based on worker capabilities.

Our experiments show that the orchestration server introduces negligible overhead, accounting for less than 0.15% of the inference time. In high-bandwidth environments, networking time remains below 25% for up to 10 workers, suggesting that execution is primarily compute-bound. Integrating token decoding into the ONNX graph reduces data transfer by 75 MB per request, while prefix caching improves Time to First Token (TTFT) by up to 30%. Although throughput is lower than local inference engines such as Ollama, we empirically demonstrate that web-based distributed inference is a viable, zero-setup solution for accessing large-scale models.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	ix
Abstract	xi
Contents	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Research Questions	2
1.3 Thesis Organization	3
2 Background	5
2.1 Large Language Models (LLMs)	5
2.2 Parallelism in Deep Learning	9
2.3 The ONNX Standard	10
2.4 Web Technologies for ML	12
3 Related Work	15
3.1 Centralized Inference Engines	15
3.2 Distributed Inference	15
3.3 Web-Based Inference	16
3.4 Gap Analysis	16
4 System Architecture	17
4.1 Design Philosophy	17
4.2 System Topology	19
4.3 Distributed Inference Strategy	20
4.4 Communication Protocol	23
5 Implementation	29
5.1 Technology Stack	29
5.2 Key Implementation Challenges	31
5.3 Server Implementation Details	36
	xiii

6	Evaluation	39
6.1	Experimental Setup	39
6.2	Baseline	42
6.3	RQ2: System Overhead	46
6.4	RQ3: Network Optimization	48
6.5	RQ4: Fault Tolerance	53
6.6	Summary of Findings	58
7	Discussion	61
7.1	Interpretation of Results	61
7.2	System Design and Trade-offs	65
7.3	Limitations and Threats to Validity	67
7.4	Future Work	69
8	Conclusion	73
	Overview of Generative AI Tools Used	75
	List of Figures	77
	List of Tables	79
	List of Algorithms	81
	Bibliography	83

Introduction

1.1 Motivation

With the advent of GPT-3 [BMR⁺20] in 2020, large language models (LLMs) have entered mainstream awareness. Since then, the computational requirements of frontier models have steadily increased. The Llama model family began in 2023, with the largest model having 65 billion parameters. In 2025, open-weight models have reached much larger parameter sizes, such as DeepSeek-V3 with 671 billion parameters or Kimi-K2 with 1 trillion parameters [Epo25].

Although it is possible to run smaller models on consumer hardware, large frontier models require hundreds of GB of VRAM. This restricts full access to such models to large institutions; individuals and smaller institutions usually lack the required computational resources. In addition, inference engines such as vLLM [vll25] are optimized for server-grade hardware and throughput rather than end-to-end latency on consumer-grade hardware.

Using distributed inference, we can pool multiple devices and run models that would otherwise be too large to fit in memory. vLLM supports scaling in this way, but only for homogeneous hardware. Others, such as Petals [BRC⁺23] or EdgeShard [ZSC⁺25], run LLM inference on heterogeneous consumer hardware and edge devices; a better fit for individuals to pool resources. However, these existing solutions introduce a usability problem. They rely on native execution environments that require manual setups and model downloads. This complexity limits participation to technical users already part of the ecosystem, preventing non-experts from exploring the newest frontier models. Furthermore, even for technically proficient people, there is an up-front time investment in setting up distributed inference with existing solutions.

A potential solution to this usability problem is the web browser. Transitioning distributed inference to the web could enable zero-setup participation, where users contribute

compute power simply by opening a website. However, realizing this vision presents technical challenges not present in native distributed systems. Browsers are sandboxed environments that prohibit high-speed memory-to-memory connections like RDMA, lack access to native CUDA kernels, and impose security constraints on GPU usage. Standard inference engines, such as vLLM, are designed for native execution and cannot operate within these constraints.

In this thesis, we address the gap between distributed inference and web accessibility. We propose a centralized architecture for distributed LLM inference using the web browser. By leveraging modern web standards, specifically WebGPU for hardware acceleration and WebSockets to communicate binary data, we design a system where the complexity of model partitioning is handled entirely by the server, requiring no installation or configuration from the user.

We explore the architectural trade-offs required to adapt LLM inference to the browser. Specifically, we investigate whether the overhead of web-based technologies such as ONNX Runtime Web, WebSockets, and Protobuf serialization is a justifiable trade-off for the gain in accessibility. Through evaluation on a dedicated testbed, we quantify the performance characteristics of this approach and determine when web-based distributed inference is a viable alternative to native solutions.

1.2 Research Questions

In this thesis, we will answer the following research questions:

Zero-Setup The web is one of the best technologies for distributing and executing code. While its primary application remains content delivery, its utility extends beyond traditional websites: In this thesis, we use it for distributed LLM inference, allowing participants to share their computational resources with the server without requiring a complex setup or manual model download. This leads us to the following research question:

How can web technologies enable zero-setup distributed LLM inference across heterogeneous consumer hardware?

System Overhead To determine when web-based distributed LLM inference is beneficial, we need to quantify the system overhead. In our experiments, we will measure end-to-end latency as we vary parameters such as model size, the number of workers, and available VRAM. We will answer:

What is the overhead of distributed LLM inference compared to local native execution?

Network Optimization LLM inference over the internet is subject to much higher latencies and lower bandwidths than inference in a data center. Therefore, optimizing networking and reducing traffic is essential for reducing end-to-end latency during inference:

To what extent do specific transport optimizations reduce transfer volume and improve inference speed in bandwidth-constrained environments?

Fault Tolerance In a web environment, participants might disconnect randomly or experience unexpected performance degradation. A system built on such workers must be fault-tolerant and handle unexpected disconnects gracefully. We will determine the breaking point of our system, giving an indication of the maximum worker volatility the system can handle:

How does system throughput degrade as worker churn rate increases?

1.3 Thesis Organization

In Chapter 2, we explain all concepts required to understand the architecture and implementation of a web-based distributed LLM inference server. We list related works in 3. Chapter 4 covers our design goals and the system’s architecture. In Chapter 5, we detail specific challenges when implementing the system. The Chapters 6 and 7 focus on our results, answering research questions through empirical evaluation. We explore limitations and future work. Lastly, Chapter 8 concludes the thesis with a summary of our contributions.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Background

This chapter provides a background for understanding web-based distributed LLM inference. We explore LLMs and their architecture, parallelism in deep learning, the ONNX standard, and machine learning in the browser.

2.1 Large Language Models (LLMs)

Through GPT-2 [RWC⁺19] and GPT-3 [BMR⁺20], LLMs have reached mainstream adoption and emerged as one of the most powerful machine learning techniques to date. Their ability to store information and synthesize text at scale is unmatched by any other technology. As such, LLMs have been at the core of machine learning research since the advent of the transformer [VSP⁺17].

To understand and build an effective LLM inference system, we will provide an overview of the transformer architecture, explore LLM inference, and list common inference challenges.

2.1.1 The Transformer Architecture

The transformer architecture [VSP⁺17] is at the core of every modern LLM. It is a neural network that excels at capturing relationships in sequential data, making transformers an ideal candidate for natural language processing.

The Attention Mechanism

The main innovation of the transformer is the attention mechanism. The input of an attention block is constructed as QKV-triples of tensors. By computing the dot product between the query (Q) and the key (K), the model captures the relationships between pairs of elements in the input sequence. For example, in the sentence *The cat sat on the*

mat because it was tired. the dot product of *cat* and *it* might be particularly high; the model learned that *it* refers to *cat*. We multiply this importance by the value (V) to get a new tensor as our output. Formally, we get:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V \quad (2.1)$$

The required computation in a transformer scales quadratically with the sequence length. For every new token, we need to calculate its relation to all previous tokens in the sequence. Improving the transformer and its attention mechanism has been extensively covered in the literature [DFE⁺22, KLZ⁺23, DLF⁺24, DLM⁺25].

2.1.2 LLM Inference

High-Level Data Flow

Model Inputs To run LLM inference, we start with the user's input text. This text is encoded as tokens, often using the Huggingface `tokenizers` library [MP23]. For example:

Tokens are a text representation that LLMs understand!

is tokenized as

[29300, 525, 264, 1467, 13042, 429, 444, 10994, 82, 3535, 0]

We can see that the input text is mapped to numbers. These tokens are called *input_ids* when running an LLM. Along with two additional tensors that are typically required, we get the following model inputs and shapes:

- *input_ids*: [batch_size, sequence_length]
- *attention_mask*: [batch_size, total_sequence_length]
- *position_ids*: [batch_size, sequence_length]

In machine learning, we can compute the output for multiple inputs in a single forward pass. The batch size refers to the number of inputs. For large-scale inference with many parallel requests, batching requests is a complex problem. Using a batch size of one is simpler and typically enough for systems that do not focus on scaling.

There are three different variants of the sequence length. This distinction becomes important when using a KV cache during inference (see Section 2.1.2). The total sequence length is the sum of all tokens processed so far. Past sequence length refers to the number of tokens that have already passed through the model. The sequence length is the difference between them, i.e., the length of tokens that will be passed through the model next. In autoregressive token generation, the sequence length is typically 1, since we generate one token at a time.

Embeddings Once the three inputs are passed to the model, they are converted to word embeddings. These embeddings are learned as higher-dimensional vector representations of individual tokens. They capture the basic meaning of individual tokens; for example, *apple* and *tree* might be closer together in this vector space than *apple* and *window*.

Transformer Layers The embeddings are passed through a sequence of transformer layers. The output of each previous layer is used as the next layer’s input, resulting in an entirely linear data flow through the model. During this process, new tokens are enriched with the context of previous tokens through the attention mechanism.

Logits The output of the last transformer layer is converted to a tensor called *logits*. Each element of this tensor has the same length as our token vocabulary, a one-hot encoding of our token ids. We can use the logits in a decoding process to find the output token of the model.

Decoding Strategies

There are many LLM decoding strategies [SYC⁺24], four common ones being [Dec25]:

- **Greedy Search** always chooses the most likely next token. We use argmax on the logits to produce the next token.
- **Beam Search** [MY17] is a strategy in which we generate our logits, choose the top-k next tokens, and run inference on all of them again. From our resulting k^2 outputs, we choose the top-k, continuing the generation process. This allows us to explore different sequences in parallel while keeping the search space small.
- **Top-k Sampling** [AMY18] uses the generated logits as a probability distribution to randomly sample a next token from the top-k tokens.
- In **Nucleus Sampling** [HBD⁺20] we choose a value $p \in [0, 1]$. We then randomly sample using the top-k tokens so that the sum of their probabilities exceeds p .

Autoregressive Generation

Using the generated token, we can run autoregressive generation: we generate one token at a time, using the previous token as input to the next model forward pass. Since we

generate one token at a time, we can set `sequence_length` to 1, drastically reducing the tensor size and computational requirements.

The KV Cache

The key-value cache (KV cache) is an optimization of the attention mechanism. Since new tokens only attend to previously generated tokens, we can cache the keys and values of already generated tokens. For a total sequence length of n , this reduces the required computation from $O(n^2)$ to $O(n)$ for each forward pass. As a trade-off, we need to store two tensors for each layer with shape `[batch_size, num_kv_heads, past_sequence_length, head_dim]`. For L layers, this results in $O(Ln)$ memory. For each subsequent forward pass, we provide the single generated token and the cache from the previous run to compute the next token.

2.1.3 Computational Challenges

VRAM Constraints

Model Name	Parameters	Full Precision	Quantized
Qwen3	0.6B	1.5 GB	500 MB
Qwen3	8B	15.2 GB	5.2 GB
Qwen3	32B	61 GB	20 GB
Qwen3	235B	438 GB	142 GB
DeepSeek-V3	671B	1.27 TB	404 GB

Table 2.1: The model size in bytes for different models. The model size for full precision was taken from Huggingface [Hug25a], for the quantized model size we used Q4_K_M GGUF from the Ollama model library [Oll].

The memory required to run an LLM can be split into three parts: model parameters, activation memory, and the KV cache. The model’s weights are loaded into memory once and remain static in size. Memory for activation is ephemeral and small; it can be ignored for calculations. The KV cache grows linearly with the sequence length as autoregressive generation continues.

Table 2.1 relates the number of model parameters to their model size in bytes. Memory requirements at the beginning of a generation are roughly equal to the model’s weights. Depending on the number of parallel inference requests and sequence lengths, memory requirements might remain small or increase significantly.

For longer sequences where KV cache growth becomes problematic, Infinite-LLM [LZP⁺24] presents an inference system for highly dynamic context lengths. With PagedAttention, vLLM [KLZ⁺23] optimizes memory layout during inference.

Using mixture of experts (MoE) models is a technique to reduce the number of parameters that need to be activated during inference, reducing computational requirements. Switch

Transformers [FZS22] scale to trillion parameter models using MoE. Similarly, many open-weight models use MoE to scale to larger parameter counts while keeping the model efficient during inference [YLY⁺25, DLM⁺25].

Memory-Bound vs. Compute-Bound Execution

The KV cache reduces computational cost during autoregressive generation. The initial prefill phase, which processes the entire input prompt, is compute-bound since it has no cached values. In contrast, the decode phase is memory-bound as it generates tokens one at a time using the cache. Since prefill has different computational requirements and can be significantly slower than decode, especially for long prompts, DistServe [ZLC⁺24] proposes running these phases on different GPUs to optimize inference efficiency.

Prefix caching is a technique in which the KV cache for the system prompt is generated ahead of time. This allows the inference engine to use the cache for the first part of the input, generating the cache only for the user's prompt.

2.1.4 Quantization

During model training, a model's weights need to have high precision in order to perform gradient descent; weights need to be updated in small steps during the backward pass. For inference, this is not necessary. Quantization [DLBZ22, FAHA23, LTT⁺24] is a technique in which the model's weights are transformed to lower precision, saving space and allowing for more efficient inference.

2.2 Parallelism in Deep Learning

Machine learning models are typically represented as dataflow graphs. The nodes are computational operators such as `MatMul` or `SoftMax`; the edges represent input and output tensors. During a forward pass, the execution engine waits for all input tensors to be present and then computes the operator. In case the model or data is too large to fit on a single device, parallelization is required [ZLZ⁺22].

Existing parallelization approaches in deep learning are typically categorized into data and model parallelism [SPP⁺20]. Within model parallelism, there are two paradigms: pipeline and operator parallelism [BOV24, ZLZ⁺22].

Data Parallelism In data parallelism, the dataset is split and distributed to multiple workers. Each worker runs a copy of the model. This method works great for small models with large data; for LLMs the opposite is true.

Pipeline Parallelism In pipeline parallelism, also referred to as inter-operator parallelism, the model is partitioned and distributed across multiple workers. Partitioning involves grouping different operators and assigning them to a worker. During a forward pass, intermediate tensors need to be transferred between workers.

GPipe [HCB⁺19] pioneered pipeline parallelism for deep neural networks, with others following shortly [NHP⁺19, YWN⁺25].

Operator Parallelism In operator parallelism, also known as intra-operator parallelism or tensor parallelism, individual operators, such as a `MatMul` operation, are split and computed on distributed hardware. For this type of parallelism to work effectively, we require fast interconnects since it requires regular synchronization between devices.

Parallelism for LLMs Each type of parallelism has seen extensive coverage for LLM training and inference: [TTHT21, HIZ⁺22, YWN⁺25] focus on pipeline parallelism, [SPP⁺20] uses operator parallelism for training LLMs at scale, and [YJK⁺22, ZLZ⁺22] combine strategies.

2.3 The ONNX Standard

In this thesis, we use ONNX [onn25] to store and serve LLMs; therefore, covering the internals and data representations of the format is essential.

ONNX represents machine learning models as a directed acyclic graph (DAG) and stores the graph along with tensor information as a binary blob encoded using Protocol Buffers [pro25].

Figure 2.1 shows the dataflow graph of a single LLM layer. We can see the attention block in the center with the KV cache as inputs and outputs. To run this layer, we would need to input the KV cache together with `output_0` and `output_3` of the previous layer. After running the layer, we get back the present KV cache along with `output_0`, and `output_3` to be used as inputs for the next layer.

2.3.1 Computational Graph Representation

In its simplest form, an ONNX file can be thought of as a DAG along with meta information. The DAG is represented as a list of nodes, each node having inputs and outputs. The graph itself also has inputs and outputs. An LLM has `input_ids`, `attention_mask`, `position_ids`, and the KV cache as the inputs; the outputs are `logits` and the updated KV cache.

The nodes are stored as a topologically sorted list. Inputs and outputs are lists of strings. The model weights are stored as initializers, which are tensor values used to specify constant inputs of the graph.

2.3.2 Tensors and Data Types

In ONNX, initializers are stored as a combination of data, shape, and datatype. Data can be stored internally or externally. The shape is represented as a list of integers. The datatype is stored as an enum.

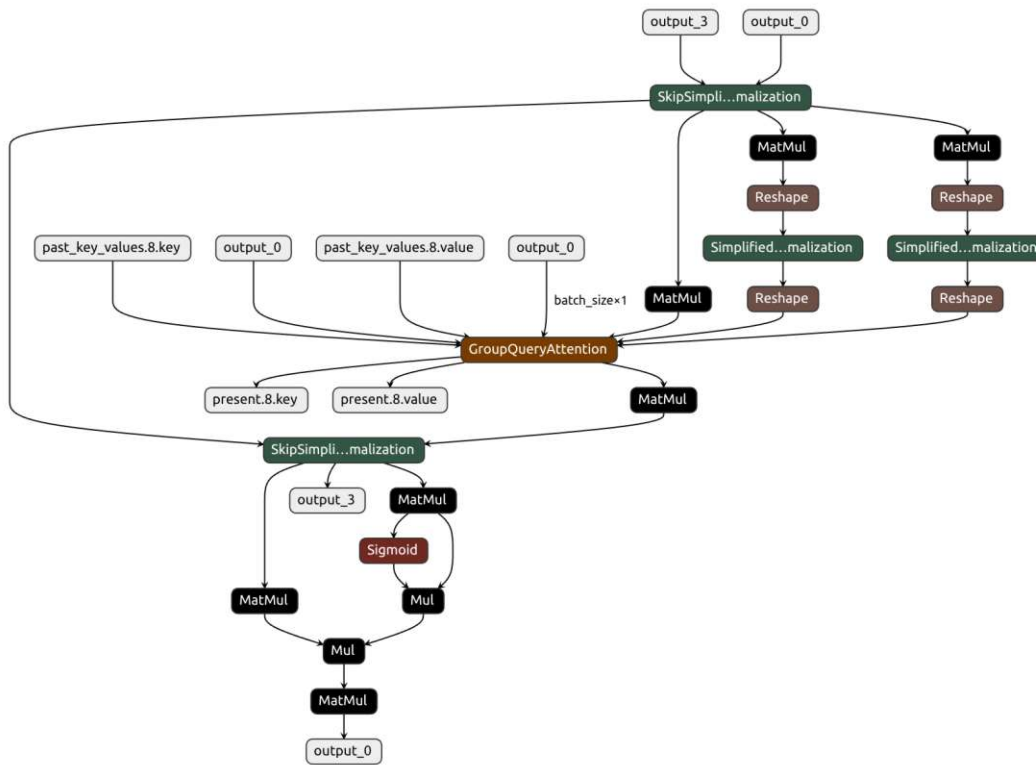


Figure 2.1: A single layer of Qwen3-0.6B stored as an ONNX file and visualized using Netron.

Tensors are stored with datatype and shape information; they also support integer or string names. String names are used for dynamic shapes, for example, the KV cache has the following shape: `float32 [batch_size, sequence_length, 1024]`.

2.3.3 Operators

The operators, such as `MatMul` or `ArgMax`, are stored in nodes as strings. The execution engine parses the nodes and computes the given operator once all node inputs are present. Since the node list is topologically sorted, the simplest way to obtain the final output is to compute the nodes in order. Inference engines typically parallelize execution when possible.

2.3.4 ONNX Runtime (ORT)

ONNX Runtime [ONN18] is an execution engine for ONNX models. The engine loads the ONNX file and computes the model's output given a map of named inputs. One of its biggest advantages is flexibility. ONNX Runtime can be used on most major platforms,

including Windows, Linux, Android, and the Web. It supports arbitrary inputs and outputs and offers an extensive selection of operators.

Execution Providers (EPs)

ONNX Runtime can be built for execution in different environments. For example, `CUDAExecutionProvider` uses the GPU with CUDA to execute the model, and `CPUExecutionProvider` executes using the CPU.

ONNX Runtime Web is a variant built for execution in web environments. The engine is compiled to WebAssembly and can be used from JavaScript as a WebAssembly module. Using WebGPU, it can execute ONNX models on the GPU.

2.3.5 Converting LLMs to ONNX

Most open weights models are available on Huggingface [Hug25a]. Since very few models are available as ONNX files, we need to convert them to use ONNX Runtime. The Python library `onnx` has an export function; however, it is impractical to use and produces unoptimized ONNX models. Optimum [hug25b] can be used to produce ONNX models using a simple command. Similarly, Olive [mic25] can produce optimized ONNX models directly from LLMs stored on Huggingface.

2.4 Web Technologies for ML

Machine learning in the web has received limited attention in the literature. [MXZ⁺19, GHA23, WJC⁺25] have investigated deep learning frameworks and browser-based inference; however, new standards such as WebGPU have not been explored in detail. WebLLM [RQZ⁺24] shows that running LLMs in the browser with WebGPU can reach speeds comparable to native inference.

2.4.1 WebGPU

WebGPU [Webb, Webd, Web25] is the successor to WebGL. It is a graphics API allowing access to the system's GPU. It is still in its infancy and under active development. As such, it is not yet enabled by default in all browsers or on all platforms.

Compute Shaders

WebGPU currently supports two types of graphics pipeline: a pipeline for rendering graphics and a compute pipeline for general computation. For LLM inference, the compute pipeline lets us use the system's GPU for inference. To use it, we need to provide WebGPU shaders written in the *WebGPU Shading Language*. Using ONNX Runtime Web, most of this work is done for us. We only need to provide the model and inputs; the engine creates and runs the pipeline and returns the outputs.

2.4.2 Browser Constraints

Sandboxing Compared to native execution, where you can run arbitrary code, the browser is more limited. It needs to protect users from malicious actors and websites. As such, we cannot execute arbitrary code; we are limited to a sandboxed environment [Chr, Sec]. As an example, finding the amount of VRAM a GPU has in Python is fairly easy; in the browser this is not possible using standard approaches. One reason for this is that exact measurements, such as the amount of VRAM in bytes, would make fingerprinting and tracking of users easier [Webc].

Memory limits WebAssembly has a memory limit of 4GB due to 32-bit addressing. Therefore, we cannot load LLMs with individual files larger than 4GB. Instead, we need to split the model’s weights so that we remain within the limit.

Performance Running natively on the GPU allows inference engines to use specialized instructions and compile models specifically for the target GPU. Since the WebGPU API needs to work for a wide variety of hardware, it has to provide an abstraction. Additionally, the browser has to ensure that shaders are correct and that a malicious actor cannot access out-of-bounds memory [Webc]. This adds overhead, so we lose performance by running in the browser instead of natively.

2.4.3 Transport Protocols

In this section, we will explore the technologies used for communication between workers and the server. In Section 4.4 we explain our choice of technologies in more detail.

WebSockets

WebSockets [MF11, The25] allow for two-way communication between a user’s browser and the server. Both sides can send and receive messages without having to poll repeatedly. Communication starts with a handshake, followed by data transfer. Data is transferred as frames, each frame being textual data, binary data, or control frames. Both sides can close the connection at any point by sending a close message. By design, WebSockets have minimal overhead, essentially exposing raw TCP to the communicators.

Protocol Buffers (Protobuf)

For communication between workers and the server, we use binary data encoded by Protobuf [pro25]. Protocol Buffers can be used to serialize structured data into bytes. By defining a common schema known to both communicators, no schema information needs to be included in the message. This makes Protobuf very efficient in communicating structured data.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Related Work

In this chapter, we will detail the latest work in the area of LLM inference. We will start with centralized inference engines, move on to distributed inference, and then cover LLM inference in the browser. Lastly, we will show that there is a distinct lack of literature using the browser for distributed inference.

3.1 Centralized Inference Engines

LLM inference is extensively covered, both in the literature and through production-ready solutions. Especially notable is vLLM [vll25], an engine originally built to showcase an improvement in the attention mechanism for faster inference called PagedAttention [KLZ⁺23]. Ollama [oll25] is built on top of llama.cpp [ggm25], running LLM inference on consumer hardware. Other inference engines include Aphrodite [aph25], SGLang [sgl25], DeepSpeed [dee25], and TensorRT LLM [NVI25].

3.2 Distributed Inference

Many centralized inference engines support distributed inference. vLLM can be run with both pipeline and tensor parallelism; SGLang supports all parallelism modes. However, these engines typically require homogeneous hardware, focusing more on throughput and server-grade hardware.

Petals [BRC⁺23] run LLMs on heterogeneous geodistributed hardware. They allow for reliable inference even if some devices randomly disconnect. Newly connected workers choose a subset of the LLM to serve. Users sending inference requests use a shortest-path algorithm to find a subset of workers for inference.

System	Distributed	Heterogeneous	Zero-Setup
vLLM	✓	✗	✗
Ollama	✗	✗	✗
Petals	✓	✓	✗
WebLLM	✗	✗	✓
Ours	✓	✓	✓

Table 3.1: Comparison of LLM inference systems

EdgeShard [ZSC⁺25] runs collaborative inference using edge devices and cloud servers. They use dynamic programming to find model partitions that optimize inference latency and throughput, taking into account heterogeneous hardware and network conditions.

Other works cover aspects such as deploying LLMs on mobile devices [ZSL⁺24], running MoE models on edge devices [YGW⁺25], or providing an open-source framework for inference on heterogeneous consumer hardware [exo25].

3.3 Web-Based Inference

Regarding web-based LLM inference, the literature is scarce. WebLLM [RQZ⁺24] compiles LLMs into WebGPU kernels using TVM [CMJ⁺18], keeping up to 80% of native performance. WeInfer [CMSL25] improves WebLLM by implementing more efficient buffer-reuse strategies and switching to an asynchronous approach that does not block GPU execution. Wllama [Ngu25] is an experimental project that provides WebAssembly bindings for llama.cpp. MediaPipe LLM by Google [LLM] offers an LLM Inference API in the browser for Gemma models. Transformers.js [Tra] offers a functional equivalent of the Python transformers library for the web. They use ONNX Runtime Web to run LLMs in the browser.

3.4 Gap Analysis

Table 3.1 compares existing systems in key dimensions for accessible distributed inference.

Existing systems make trade-offs that limit accessibility. vLLM and SGLang require homogeneous hardware and a complex setup. Petals supports heterogeneous distribution, but requires manual model downloads and installation of Python dependencies. WebLLM runs in the browser, but cannot distribute computation.

Our system combines web-based execution with distributed inference for heterogeneous workers. This trade-off prioritizes accessibility over raw performance: web-based execution is slower than native execution, and distributed coordination adds latency. However, these costs enable deployment scenarios that are inaccessible with existing systems: users can pool their resources to run models too large for any single device, without requiring software installation or homogeneous hardware.

System Architecture

This chapter provides a high-level overview of our architecture. We will start by explaining our goals and non-goals. This is the foundation upon which further assumptions and trade-offs are built. Next, we will focus on the system's topology and core components. In Section 4.3, we investigate trade-offs when designing a distributed inference system, showing why we opted for pipeline parallelism and a distributed KV cache without synchronization. Lastly, we explain the communication protocol.

4.1 Design Philosophy

In this section, we describe our main design objectives. Clear goals allow us to make better assumptions upon which further analysis is built.

4.1.1 Goals

Zero-Setup Distributed Computing

The central idea of our architecture is to reduce setup time by running inference in the browser. As detailed in Chapter 3, there are solutions for running large LLMs on distributed consumer-grade hardware; however, they require manual setup. We choose the browser as our primary computing platform. This decision represents a trade-off between user-facing complexity and inference speed. Communication over the internet and execution in the browser using WebGPU are not as efficient as running natively with high-speed interconnects.

Flexibility

Although our main design goal is to support web-based workers, our system allows any worker to connect and contribute as long as they implement the communication protocol

correctly. This allows us to connect workers running natively.

Worker Heterogeneity

Workers should be able to run on diverse hardware and software, including CPUs, WebGPU, and native GPUs. It should be possible to use a highly optimized custom ONNX inference engine on server-grade hardware. Workers have different available VRAM and execution speeds depending on their exact configuration. Workers might run on different networks or have differing latencies and bandwidths. Our system is designed to use any such worker as long as they correctly implement the communication protocol.

Fault Tolerance

Since communication uses WebSockets over the internet, there is some inherent instability. The inference server must handle varying network conditions, varying execution speeds, and worker degradation. Users and workers might also disconnect randomly.

Latency

Our main optimization goal is end-to-end latency during token generation. We optimize for efficient single-request inference, rather than multi-user scenarios.

4.1.2 Non-Goals

Throughput

Under the assumption that we have enough resources, distributed inference cannot be as efficient as centralized inference due to communication overhead. This problem is even more pronounced for distributed inference over the internet using consumer hardware. Our goal is not to compete with optimized high-throughput solutions such as vLLM, but to demonstrate that the web is a viable platform for distributed LLM inference. We prioritize running large models on consumer hardware in single-user environments over scalability.

Privacy and Security

To run autoregressive token generation, a worker needs to have full information about its inputs and outputs. The worker could fetch and reconstruct the remaining model and complete the user's inference request, providing the required information. We emphasize that privacy-related concerns are not within the scope of this thesis.

Regarding security, we assume full cooperation of all components without malicious intent; otherwise, we would need to verify their output. This could be done by running the same computation on multiple workers and comparing their results; however, we would need to invest additional resources. Our goal is to design a system for individuals and small institutions, so we assume that all components are known and trusted.

4.2 System Topology

4.2.1 Star Topology

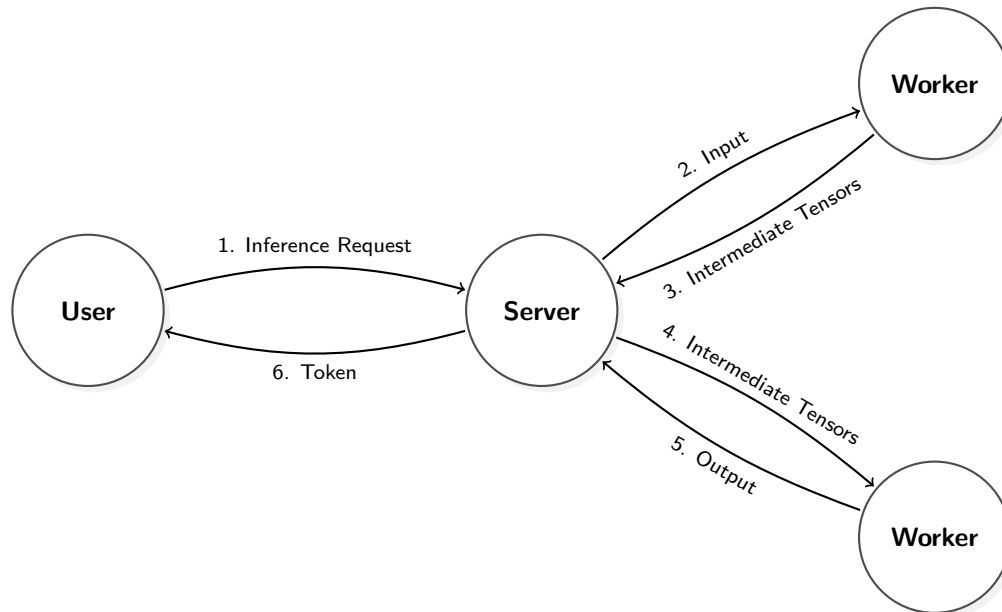


Figure 4.1: The system’s star topology during inference, illustrating the centralized coordination of the server.

As shown in Figure 4.1, the system is designed as a star topology. The inference server is at the center, with workers and users connecting to it. All communication flows through the server; users and workers do not communicate with each other. This architectural pattern is a natural choice for a web-based approach with transient workers; a central server is required to serve the website. By extending the server to run inference, we get a single point of coordination and metric collection, simplifying state management.

4.2.2 Components

Inference Server

The core component of our system is the centralized inference server. It has the following responsibilities:

- **API:** The server needs to provide the API for users and workers. Users send inference requests and expect back a stream of generated tokens. Workers connect and share their computational resources with the server.
- **Metric collection:** To effectively partition the model, the server collects metrics about the ONNX model and the workers.

- **Partitioning:** Using the collected metrics, the server has to partition the ONNX model such that end-to-end latency is minimized.
- **Orchestration:** Given an assignment of model parts to workers, the server is responsible for coordinating token generation. It needs to assign model parts to workers and determine when all workers have successfully loaded their models. For subsequent inference requests, it needs to orchestrate autoregressive token generation. Section 4.3.2 describes distributed token generation in detail.

Compute Worker

Workers use the server’s API to join the system. They have the following responsibilities:

- **Offer Computational Resources:** The worker’s main responsibility is to execute workloads upon the server’s instruction. The worker should do so as soon as requested and to the best of its ability.
- **Metrics:** While connected, the server and worker collect metrics. The worker must provide accurate measurements of the bandwidth and computational speed.
- **KV Cache Management:** Workers manage their own KV cache as described in Section 4.3.3. During inference, the workers use the KV cache to generate tokens.

User

Users connect to the server and submit inference requests. They expect a stream of decoded tokens from the server.

4.3 Distributed Inference Strategy

4.3.1 Pipeline Parallelism

As detailed in Section 2.2, parallelism in deep learning can be categorized into data, pipeline, and tensor parallelism. Our goal is to run LLMs that are too large for a single device using distributed hardware. Data parallelism does not alleviate memory or compute requirements; it just allows for higher throughput. The choice is therefore between pipeline and operator parallelism.

Tensor parallelism requires regular synchronization. For matrix multiplications, workers must perform an AllReduce to combine partial results. In transformer models, this synchronization occurs multiple times per layer. To simplify, we assume that we need to synchronize once per layer. Consider a model with $L = 32$ layers. With $N = 4$ workers at 10ms round trip time (RTT), end-to-end network time T becomes:

$$T_{\text{operator}} = L \times t_{\text{sync}} = 32 \times 10\text{ms} = 320\text{ms per token}$$

This calculation assumes perfect parallelism; in a real-world scenario, we would need to wait for the slowest worker, so the RTT increases to the maximum across all workers. Since inference speed is typically measured in tokens per second, we can see that this is not sufficient.

Pipeline parallelism does not require synchronization at layers. Instead, we split the model into chunks, computing each part in sequence and forwarding the result to the next worker. For the same 32 layers and 4 workers, we get a total network latency of:

$$T_{\text{pipeline}} = N \times t_{\text{transfer}} = 4 \times 10\text{ms} = 40\text{ms per token}$$

This represents an 8x improvement in network latency. Tensor parallelism also requires transferring much more data. The trade-off is reduced parallelism and idle workers. Given our focus on end-to-end latency, as detailed in Section 4.1, we adopt pipeline parallelism over tensor parallelism. By splitting at layer boundaries, we can formulate partitioning as a linear assignment problem: given L layers and N workers, assign each layer to exactly one worker to minimize end-to-end latency. The partitioning problem is explored in detail in Section 5.2.1.

4.3.2 Inference Data Flow

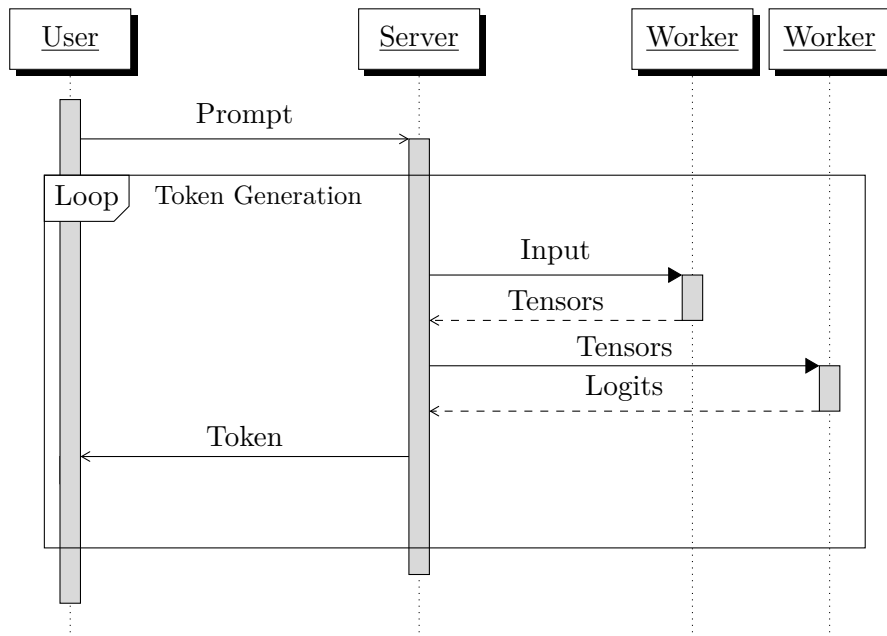


Figure 4.2: Distributed Autoregressive Token Generation

We established that using pipeline parallelism works best for our inference goals. In this section, we will detail the data flow during autoregressive token generation. Figure 4.2 shows a sequence diagram for generation using two workers.

We start with a user submitting a prompt. The server tokenizes the prompt and creates the input ids, attention mask, and position ids. These tensors are sent to the first worker. The worker computes a forward pass and returns the result to the server, which stores all intermediate tensors. This is necessary since some output tensors are used in multiple future layers. The server then collects and sends all required tensors to the next worker. This process repeats until the forward pass is complete and the server has the final output. The output is used to compute the next token. From here on, we get autoregressive token generation where the system computes one token at a time until the end-of-sequence (EOS) token is reached. Each generated token is streamed back to the user.

4.3.3 Distributed State Management (The KV Cache)

As explained in Section 2.1.2, the KV cache is essential for efficient token generation. For distributed inference, we can choose between two main approaches: either synchronize the cache with the server or recompute it when a worker disconnects. This represents a trade-off between performance and fault-tolerance.

Let us assume that we are using Qwen3-0.6B for inference. If we choose to recompute the KV cache for a disconnect, we get the following calculation:

We need to transfer two main intermediate tensors between layers, both having the shape `[batch_size, sequence_length, 1024]`. Assuming a batch size of 1 and a sequence length of 128, each transfer requires exactly 1 MiB for 32-bit floats. This scales linearly with the number of workers, so for 2 workers this would be 1 MiB; for 10 workers it would be 9 MiB.

If we sync the cache for every generated token, we need to transfer extra bytes on each request. Qwen3-0.6B has 28 layers with a KV cache shape of `[batch_size, 8, past_sequence_length, 128]`. Since we only need to transfer the cache for newly generated tokens, we can assume a batch size of 1 and a past sequence length of 1 for autoregressive generation. This results in 224 KiB transferred per generated token. Additionally, we need to send the KV cache from the server to the workers on disconnect. The full KV cache is 28 MiB assuming a past sequence length of 128 at the time of disconnect; however, we only need to send the KV cache to workers who are assigned new layers. In addition, the size does not scale with the number of workers.

We can see that the decision of whether to sync the cache or recompute it depends on a variety of factors. Tables 4.1 show this trade-off. For small models and lower numbers of workers, recomputing the cache is often strictly better. Under the assumption that a disconnect happens eventually during generation, synchronizing the cache is strictly better for models requiring many workers or models with specially optimized attention mechanism such as DeepSeek-V3 with Multi-Head Latent Attention. Our assumption is that workers can disconnect at any point; however, we also assume that disconnects happen infrequently, and most inference requests finish without a disconnect. Therefore, we choose to recompute the cache on disconnects, focusing on generation speed. For

Model	Workers	Initial Size	Slope(Rec)	Slope(Sync)	Equilibrium
Qwen3-0.6B	1	28 MiB	0 B	448 KiB	<i>Recompute</i>
Qwen3-0.6B	2	28 MiB	8 KiB	336 KiB	<i>Recompute</i>
Qwen3-8B	4	36 MiB	96 KiB	360 KiB	11.6
Qwen3-32B	8	64 MiB	280 KiB	576 KiB	93.4
Qwen3-235B-A22B	8	47 MiB	224 KiB	423 KiB	113.8
Qwen3-235B-A22B	16	47 MiB	480 KiB	399.5 KiB	<i>Sync</i>
Qwen3-235B-A22B	32	47 MiB	992 KiB	387.75 KiB	<i>Sync</i>
DeepSeek-V3	64	17.16 MiB	3.45 MiB	139.39 KiB	<i>Sync</i>

(a) Equilibrium with a short prompt ($P = 128$).

Model	Workers	Initial Size	Slope(Rec)	Slope(Sync)	Equilibrium
Qwen3-0.6B	1	896 MiB	0 B	448 KiB	<i>Recompute</i>
Qwen3-8B	4	1.12 GiB	96 KiB	360 KiB	372.4
Qwen3-32B	8	2 GiB	280 KiB	576 KiB	2989.0
Qwen3-235B-A22B	32	1.47 GiB	992 KiB	387.75 KiB	<i>Sync</i>
DeepSeek-V3	64	549 MiB	3.45 MiB	139.39 KiB	<i>Sync</i>

(b) Equilibrium with a long prompt ($P = 4096$).

Table 4.1: Comparison of bandwidth costs between KV cache recomputation and synchronization. **Initial Size** refers to the KV cache size of the prompt. **Slope(Rec)** denotes the network transfer cost of activations between workers per generated token. **Slope(Sync)** denotes the cost per token to synchronize the new KV cache slice and the recovery cost for transferring the cache to a replacement worker. **Equilibrium** is the sequence length x where the recomputation cost ($k_{rec}x + d_{rec}$) equals the total synchronization cost ($k_{sync}x + d_{prompt}$). Values of *Recompute* or *Sync* indicate strict superiority of a strategy. *Note:* In practice, if generation finishes before equilibrium is reached, recomputation is superior as it avoids the continuous overhead of synchronization. Small tensors and model I/O are excluded; logits are optimized via elimination (see Section 5.2.2).

use cases where long sequences are required with regular disconnects during generation, synchronizing the cache is the better choice.

4.4 Communication Protocol

In this section, the communication protocol between the server, workers, and users will be detailed.

4.4.1 Transport Layer

We use WebSockets and HTTP to handle all communication. HTTP is used to download model weights from the server and to load static information, such as the names of model weights for specific layers. For all other communication, WebSockets are used.

HTTP for Weights and Static State

Using HTTP for model weights and static state enables us to leverage a mature ecosystem, including browser caching, compression, and CDN distribution. By using dynamic URLs for the model weights, we can store them on multiple servers, allowing downloads from each without blocking the inference server’s bandwidth. This becomes especially important when we serve large models with hundreds of GBs of model weights. Additionally, we avoid head-of-line blocking in WebSockets and can request multiple static files at once.

WebSockets for Control and Small Tensors

The choice for WebSockets is driven by three main requirements:

- **Browser Support:** We require a communication protocol that workers running in browsers can use. The main choices for browser-based protocols are HTTP, WebSockets, WebRTC, and SSE.
- **Bidirectional Communication:** When assigning computations, the server has to assign tasks to the worker. In response, the worker has to send back the result. For long-running tasks such as the server instructing a worker to download weights and the worker responding later, the protocol needs to support asynchronous, bidirectional communication.
- **Disconnect Detection:** The server should detect worker disconnects as fast as possible so it can continue with token generation with minimal downtime.

WebSockets support all of these requirements. They also support binary data, which simplifies the transport of tensors. A combination of WebSockets for control signals and WebRTC for peer-to-peer connection between workers is also promising. This setup avoids transferring intermediate tensors to the server. However, setup and implementation are more complex, especially when allowing native workers as detailed in Section 4.1. We leave the design and implementation of such a system for future work 7.4.2.

4.4.2 Protocol Specification

Our protocol follows a request-response pattern. State-changing messages (*PrepareModel*, *CommitModel*) require explicit responses (*PrepareDone*, *CommitDone*). This ensures that the server knows when workers are ready without having to repeatedly poll the worker’s state. Workers are reactive; they respond to server commands but do not initiate

state changes. This simplifies coordination, as the server is a single authoritative source. Figure 4.3 shows the message sequence for a worker’s lifecycle.

WebSocket communication for workers can be categorized into two main parts. Control messages are lightweight, measuring in at a few bytes. They are used to communicate state changes. Data messages contain tensors used for computations, ranging from kilobytes to a few megabytes, depending on the model’s tensor shapes.

Server Control Messages

Once a worker connects, communication starts with the server sending an *Init* message.

- **Init:** This message contains information about the shared state between the server and the worker. The server expects back a *Connect* message as a response.
- **DownloadLayers:** At any point, the server can instruct a worker to download a set of layers. This might be for redundancy or to prepare for a model assignment.
- **PrepareModel:** When the server decides to assign a model partition to a worker, it communicates the required layers and the ONNX model’s URL on the server.
- **InvalidateCache:** This message is used to invalidate the worker’s KV cache for specific inference requests.
- **ReleaseSession:** Releasing a session instructs the worker to delete the ONNX Runtime session and invalidate the entire KV cache. Until a new model is assigned, no further computation requests will be sent.

Worker Control Messages

- **Connect:** After connecting, the worker determines its own capabilities and communicates them to the server. After sending this message, the worker idles and waits for further instructions from the server.
- **LayerDownload:** After every layer download, the worker indicates which layer it downloaded and how long it took.
- **PrepareDone:** Once the worker has loaded all layers and is ready to load the model, it sends *PrepareDone*.
- **CommitDone:** This message is used to communicate that the worker has loaded the ONNX model and is ready to run workloads.

Data Messages

There are three data messages:

- **Computation:** This message is used to send tensors to a worker. Data is stored as a map from strings to tensors, matching the input format of ONNX Runtime.
- **ComputationResult:** This message contains the result of the computation along with the workers execution time.
- **CommitModel:** This message indicates that the worker should load the model into memory. When assigning a model to a worker, the prefix cache will be included in this message.

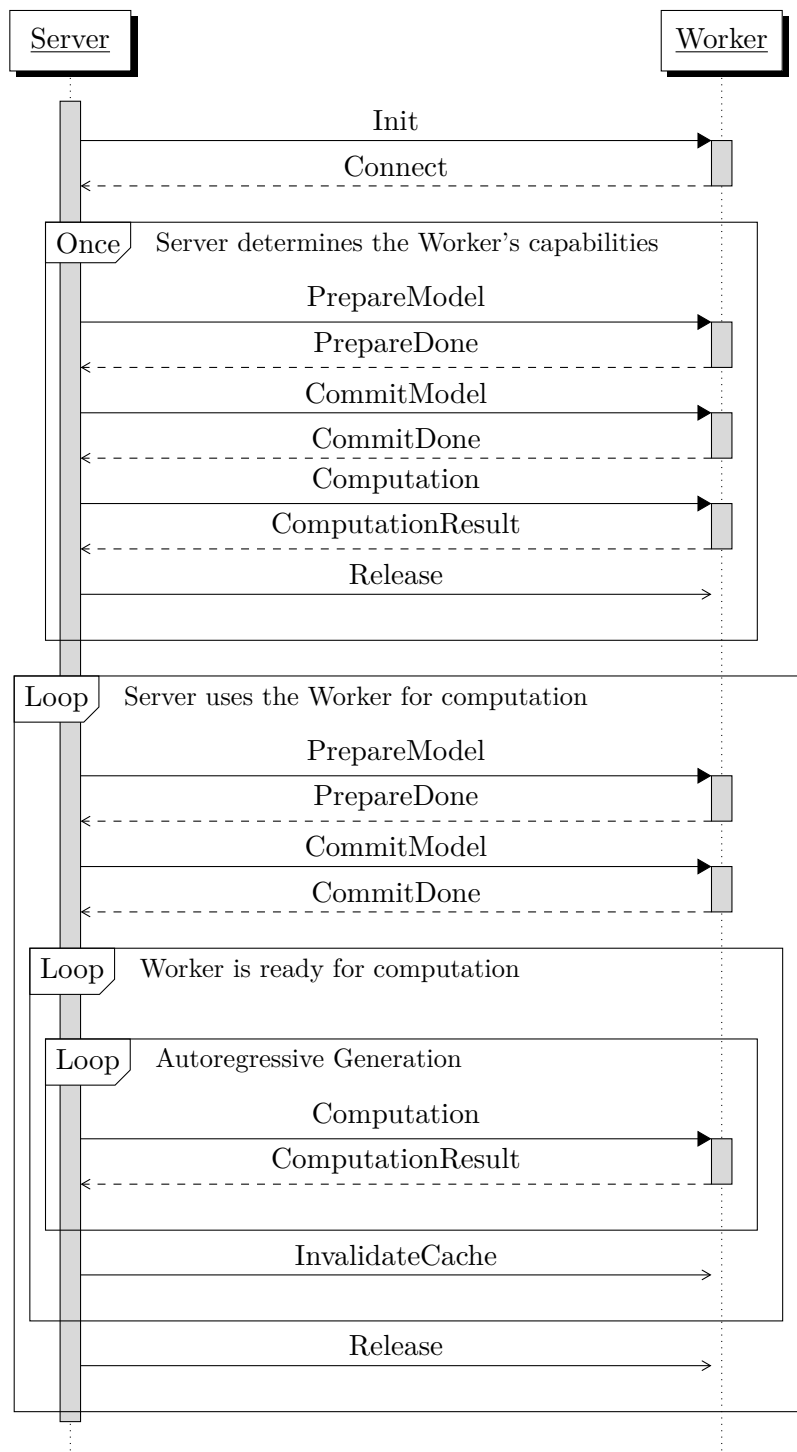


Figure 4.3: The minimum required communication for token generation from the server's perspective.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Implementation

This chapter starts with an overview of the system’s technology stack. Next, we will focus on key challenges and their solutions. Lastly, we will cover additional implementation details such as the server’s concurrency model.

5.1 Technology Stack

For our implementation, we use the following technologies:

- **Rust:** The server is written in the Rust programming language. This choice was made for three main reasons: we need to handle multiple concurrent requests that access the same state and Rust provides excellent concurrency primitives. For parsing ONNX models and tensors, we need low-level byte manipulation. Lastly, using Rust’s type system and borrow checker, we can focus on building robust abstractions and refining the server’s inference logic.
- **ONNX & ONNX Runtime:** We represent our LLMs using ONNX. Using the internal graph representation of ONNX we can quickly transform and partition the model. Since ONNX uses Protobuf, we get an efficient representation of the split model in binary. ONNX Runtime emerged as a great choice since we need to run non-standard models in the web. Other solutions such as [CMJ⁺18] TVM compile LLMs to WebGPU [RQZ⁺24]; however, they do not provide the same flexibility and ease of use as ONNX Runtime.
- **TypeScript & WebGPU:** Workers running in the browser are implemented in TypeScript and run using WebGPU. WebGPU is the best choice for running models on the GPU in the browser. WebNN [Weba] was considered; however, it is highly experimental and unstable.

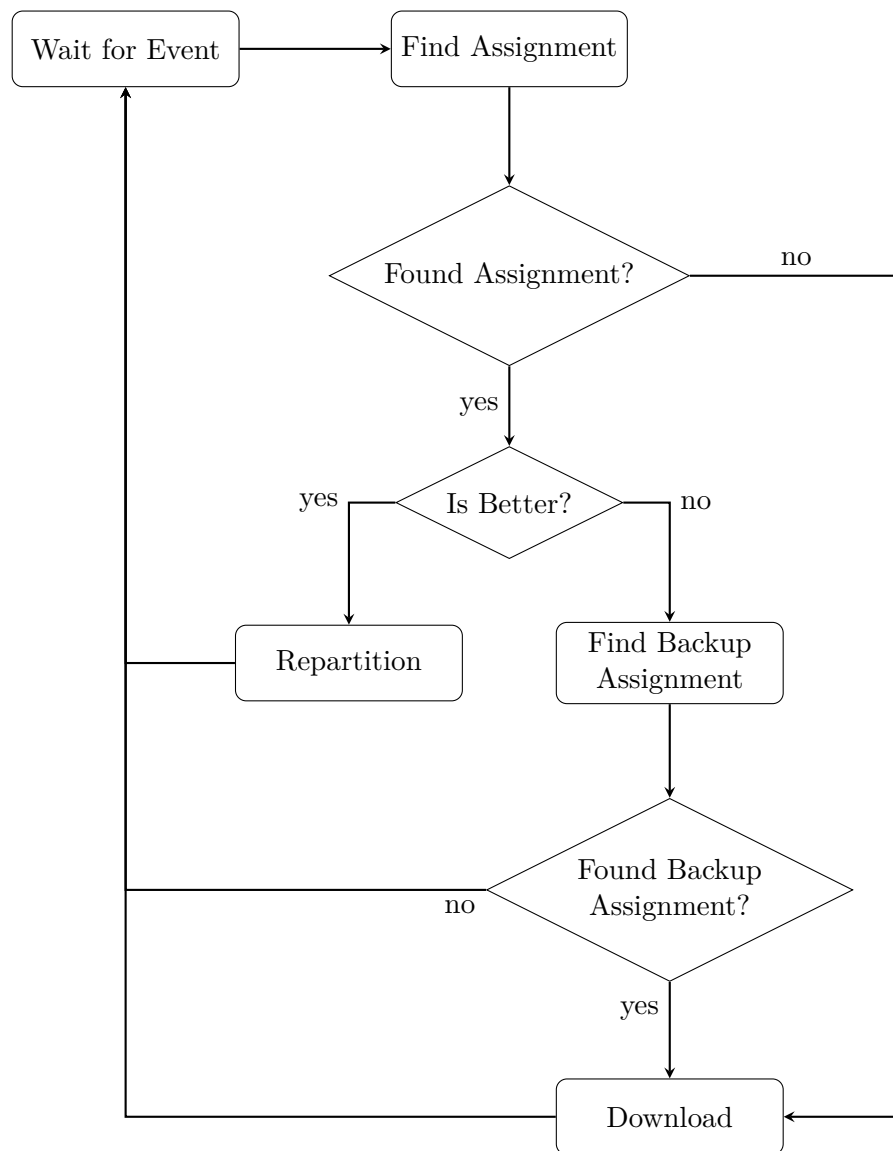


Figure 5.1: Control-flow diagram of the dynamic repartitioning loop.

- **Python & CUDA:** For native workers, we use Python and CUDA. Python is deeply integrated into the machine learning ecosystem and provides excellent support for ONNX. Our workers have a simple structure, making Python perfect for development. For our experiments, we use NVIDIA GPUs making CUDA an obvious choice.

5.2 Key Implementation Challenges

5.2.1 Challenge 1: Heterogeneous Worker Coordination

Problem

As described in Section 4.1, workers can have different configurations. The inference server is required to make partitioning decisions and run efficient autoregressive token generation using these workers. To use workers to their maximum potential, we cannot use static model partitions. It is infeasible to precompute assignments for all possible worker configurations, and assigning individual layers to workers is inefficient. We need to dynamically partition the model based on the workers capabilities.

Solution: Metrics-Based Dynamic Partitioning

Metric Collection To effectively partition the LLM, the server needs to collect metrics about the model and workers. At startup, the server loads the ONNX file and collects metrics such as weights in bytes or input tensor size for each layer. It also runs each layer a few times to get an estimate of the computational cost. By running each layer independently and freeing the resources afterward, the server can run the full model, albeit slowly due to memory offloading and filesystem overhead.

Regarding workers, bandwidth is collected by instructing the worker to download random data over 5 seconds and measuring the bytes downloaded. Latency is collected using WebSocket pings. After connecting, the worker determines its own available VRAM and the layers already downloaded. This is communicated to the server. To determine the computational capabilities of the worker, a random layer is assigned to the worker after connecting. By running this layer with dummy data, the server determines the worker's execution speed. This can be seen in Figure 4.3.

Dynamic Repartitioning In Figure 5.1, we can observe the dynamic repartitioning loop. The server waits for one of the following events: a new worker connecting, an active worker disconnecting, or a periodic timeout. Following this, the server attempts to find an assignment of continuous layers to connected workers such that VRAM is not exceeded and end-to-end latency is minimized. If no such assignment was found, the server cannot run inference. Instead, a partial assignment is used to download layers and prepare for a future assignment. If an assignment was found, it is compared with the current assignment. When comparing assignments, we split the cost into execution cost and initialization cost. The execution cost is an estimate of end-to-end latency in milliseconds for a single model forward pass during token generation. The initialization cost is an estimate of the repartitioning time. For a new assignment to be accepted, it has to fulfill:

$$C_{\text{exec}}^{\text{new}} + \frac{C_{\text{init}}}{N(t)} < (1 - \alpha)C_{\text{exec}}^{\text{current}}$$

where $N(t)$ is an amortization factor that grows with the time since last repartitioning. This factor reduces the weight of the initialization cost for long-running stable assignments, resulting in the server eventually accepting better solutions regardless of the initialization cost. α is a threshold used to only allow new assignments that are better by some margin. During our experiments, we set $\alpha = 0.2$; the server only accepts new assignments with end-to-end latency that is at least 20% lower.

Algorithm 5.1: Dynamic Programming for Layer-to-Worker Assignment

Input: Cost calculator \mathcal{C} , number of layers L , workers $\mathcal{W} = \{w_1, \dots, w_m\}$
Output: Assignment \mathcal{A} of layer ranges to workers
 // $dp[\ell][i]$ = minimum cost to assign first ℓ layers to first i workers

```

1  $dp[\ell][i] \leftarrow \infty \quad \forall \ell \in [0, L], i \in [0, m];$ 
2  $dp[0][0] \leftarrow 0;$ 
3 for  $\ell \leftarrow 0$  to  $L - 1$  do
4   for  $i \leftarrow 0$  to  $m - 1$  do
5     if  $dp[\ell][i] = \infty$  then
6       continue;
7     end
8     // Option 1: Skip worker  $w_{i+1}$ 
9      $dp[\ell][i + 1] \leftarrow \min(dp[\ell][i + 1], dp[\ell][i]);$ 
10    // Option 2: Assign layers  $[\ell, \ell')$  to worker  $w_{i+1}$ 
11    for  $\ell' \leftarrow \ell + 1$  to  $L$  do
12       $c \leftarrow \mathcal{C}(w_{i+1}, \ell, \ell');$ 
13      if  $c = \infty$  then
14        break // VRAM constraint violated
15      end
16       $dp[\ell'][i + 1] \leftarrow \min(dp[\ell'][i + 1], dp[\ell][i] + c);$ 
17    end
18  end
19 // Find furthest reachable layer
20  $\ell^* \leftarrow \max\{\ell : dp[\ell][m] < \infty\};$ 
21 // Backtrack to reconstruct assignment
22 Reconstruct  $\mathcal{A}$  by backtracking from  $dp[\ell^*][m];$ 
23 return  $\mathcal{A}$ 
  
```

Assignment Algorithm For L layers and m workers, algorithm 5.1 solves the partitioning problem for a fixed sequence of workers in $\mathcal{O}(L^2m)$ time. Because workers are heterogeneous, the assignment cost is dependent on the order of the workers in the sequence. Finding the global optimum requires evaluating all worker permutations; an NP-hard problem.

To balance optimality with solving time, we implement a hybrid optimization strategy based on the problem size:

1. **Small Scale** ($m < 8$): We evaluate all possible worker permutations to ensure we find a global optimum.
2. **Medium Scale**: We use Simulated Annealing (SA) to approximate the optimal worker orderings. The SA algorithm explores different worker ordering and uses the DP algorithm's costs to evaluate each state.
3. **Large Scale**: We employ a time-bounded random search, evaluating random worker shuffles until a strict time budget is exhausted to prevent blocking the inference server.

5.2.2 Challenge 2: Efficient Tensor Transfer

Problem

As detailed in Section 4.4, our communication consists of control and data messages. Control messages are inherently small; however, data messages typically have tensor sizes starting with a few KB up to a few MB. Some tensors, such as those required during prefill (see Section 2.1.3), can be even larger. To allow for token generation on the order of tokens per second, we need to reduce the amount of bytes transferred as much as possible.

Solutions

Binary Serialization The first optimization is to use a binary serialization format instead of a text based one like JSON. Using JSON for floating point data requires encoding every decimal as its own character. If we encode data as raw bytes, we only need four bytes per 32-bit floating point number.

We decided to use Protocol Buffers for binary encoding. ONNX is encoded as Protobuf as well; this means that we do not need to add an additional dependency. Additionally, we can encode all of our messages as binary data and transfer them over WebSockets as raw bytes. Since the schema is shared on both ends, we get type safety for communication.

Compression The weights workers need to download can easily reach multiple GB of data per worker. Depending on bandwidth, downloading can take a long time. We use Zstandard [fac25] to compress the model's weights ahead of time and reduce the amount of bytes required when downloading. For browser clients, no additional implementation is required, as browsers can automatically decompress zstd. Using zstd with a compression level of 6 reduces the models' weights by 2-3 times. Additionally, we compress Protobuf messages containing tensors during transfer using zstd with the default compression level of 3, a good middle-ground between speed and compression ratio.

Logit Elimination The largest tensors to be transferred are the model’s logits. For Qwen3:0.6B logits are of shape `[batch_size, sequence_length, 151936]`. For 32-bit floating point numbers, this results in roughly 500KB for a batch size and sequence length of 1. For generating the first token, this size is much larger since it scales linearly with the sequence length.

We remove this entire tensor by computing the next token on a worker. By modifying the ONNX graph and adding an `ArgMax` node after the logits, we can do greedy sampling directly inside ONNX, only returning a single i64 integer. This is a trade-off: we lose the option of using more advanced decoding strategies, instead reducing the amount of bytes transferred by roughly 500KB per generated token.

Content-Addressed Caching Caching the models’ weights after downloading allows us to reuse them if a worker reconnects later. We optimize model download and caching in two steps: by storing weights with their SHA-256 hash as the filename, workers can check whether they have already downloaded the weights. Secondly, once a worker has downloaded all the weights for a layer, it stores them. When reconnecting, the worker communicates which layer it has already downloaded, allowing the server to assign layers so that the worker needs to download as little new data as possible.

Prefix KV Caching Computing the first token is computationally intensive and requires transferring large intermediate tensors. We use prefix caching to generate the system prompt’s KV cache ahead of time. We transfer the prefix cache to workers once when committing a new model. This reduces the computation time and the number of bytes transferred for the first token.

5.2.3 Challenge 3: Browser Environment Constraints

Problem

As described in Section 2.4.2, executing GPU workloads in the browser is subject to limitations. In this section, we will describe how we navigated these constraints and the limitations that remain.

Solutions

Chunked Model Storage WebAssembly has a 4GB memory limit. Since ONNX Runtime is loaded as a WebAssembly module, this limit also applies to models. Instead of storing the data of the partitioned models in a single file, we write each initializer to a separate file. This allows us to run models with a total weight exceeding 4GB.

Unsafe WebGPU Although WebGPU is available in some settings, such as Chrome on Windows or Android, there are configurations where we need to run with experimental flags. For example, to use WebGPU in Chrome on Linux, we need to run with the

`-enable-unsafe-webgpu` flag. [Imp] details the implementation status of WebGPU across browsers and platforms.

Hacky VRAM Detection There is no browser API to get the exact amount of VRAM a system has. We work around this limitation by manually allocating buffers on the GPU until we run out of memory. We can only do this in 1GB chunks, since WebGPU does not allow larger allocations at a time. This means that to detect the amount of VRAM on a system, we might take some time and use all the system’s VRAM in the process. As a result, we can measure the amount of available VRAM within 1GB. As an alternative, we offer worker the option to manually report their available memory.

Precision Limitations For our evaluations, we use Puppeteer to run web workers on our servers. On desktop clients, 16-bit floating-point numbers (half precision) are generally supported. For headless clients in Puppeteer, we were unable to add the required `shader-16` feature. For a fair comparison, we defer the evaluation with reduced precision, including quantization of the intermediate tensors and the KV cache, to future work.

5.2.4 Challenge 4: Fault-Tolerant State Management

Problem

The inference server must handle random user and worker disconnects. If a worker disconnects during token generation, the server needs to quickly resume generation. We need to quickly detect disconnects or performance degradation, redistribute the model accordingly, update the distributed KV cache on all active workers, and continue generation.

Solutions

Failure Detection Since we are using WebSockets, we can detect disconnects by waiting for a close message from the socket. In rare cases, a worker can disconnect without the server’s socket receiving a close message. For these cases, we use periodic WebSocket pings. If a ping fails, we assume that the worker has disconnected. As a secondary objective, we need to detect performance degradation. WebSocket pings serve as a measure of latency; to detect degradation in computational speed, we use runtime metrics.

Preemptive Layer Redundancy To avoid long wait times on disconnect due to new workers needing to download model weights, we use preemptive layer redundancy. In Figure 5.1, we can see the repartitioning loop. We periodically attempt to find a better assignment. In case we have not found a better assignment, we try to find a backup assignment. We do this by modifying the cost function of our dynamic programming algorithm: first, by using only workers not yet assigned, and, if that fails, by prioritizing

unused workers and disallowing layer overlaps. Given this backup assignment, we instruct workers to download the layers in the background. If a worker fails, we can immediately reassign the model without downloading multiple GB of data, and then resume generation.

Fast Repartitioning Upon disconnect, the server immediately pauses token generation and attempts to find a new assignment. If such an assignment is found, this is communicated to the workers, and token generation is resumed immediately. In case there are not enough workers, the server waits for events such as a worker connecting before attempting to find an assignment.

KV Cache Recovery As detailed in Section 4.3.3, we need to recompute the KV cache upon disconnect. The implementation is simple: We store all generated tokens on the server. When a worker disconnects, we treat all active inference requests as new requests.

5.3 Server Implementation Details

5.3.1 Lifecycle

The server follows four main stages in its lifecycle in order to fulfill its responsibilities:

- **Initialization:** The server lifecycle starts with the initialization phase. It needs to load the provided ONNX model, collect model metrics, and start the web server so that users and workers can connect.
- **Discovery:** After initialization, the server waits for workers to connect. The main priority during this phase is to discover the available memory of each connected worker. Once enough total memory is available to load the full model, the server can proceed.
- **Ready:** After the model is partitioned and assigned to the connected workers, the server enters a ready state, being able to serve requests. In this state, the main goal is to efficiently serve inference requests.
- **Recovery:** If the server loses the ability to run a full forward pass due to a worker disconnect, it enters the recovery phase. During this phase, it attempts to find a new model assignment for the workers.

5.3.2 Event Loop & Server State

The server's lifecycle is managed by an event loop that distributes model layers to workers. The loop transitions between four primary states, as illustrated in Figure 5.2:

- **Down:** The server is idle or cannot find a valid assignment of model parts to available workers.

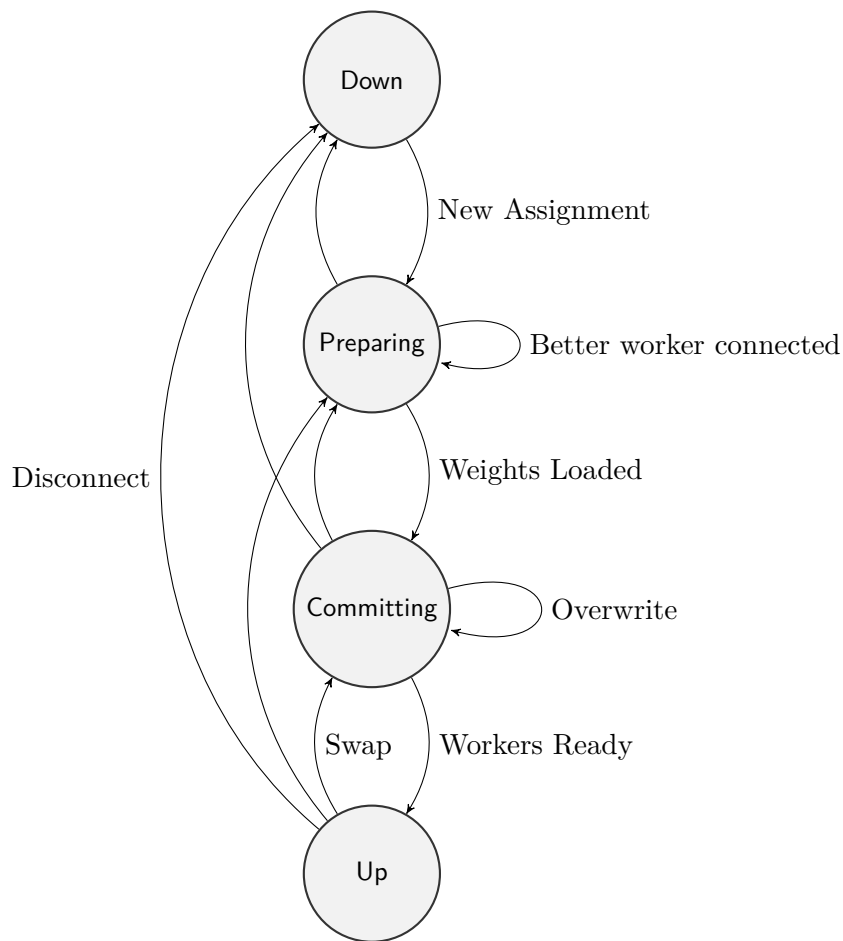


Figure 5.2: Event loop state transitions. The server uses a background staging variable (`inactive_graph`) to allow transitions from *Up* directly to *Committing* (Hot Swap), minimizing downtime during model updates.

- **Preparing:** An assignment has been generated. The server instructs workers to download layers and prepare to start the inference session. Then, it awaits confirmation from the workers.
- **Committing:** After all workers responded with *PrepareDone*, the server instructs all workers to initialize their inference sessions. During this phase, the server waits for a *CommitDone* signal from each worker.
- **Up:** The inference session is active and the server is processing inference requests.

To optimize runtime efficiency, the server maintains two state variables: `active_graph`, which represents the model currently serving requests, and `inactive_graph`, which serves as a staging area for new assignments.

This separation facilitates a "Hot Swap" mechanism. When a new worker connects, or a better partitioning strategy is found, the new assignment is loaded into the `inactive_graph`. The server initiates the *Preparing* phase for this background graph without interrupting the currently active session.

Once the workers associated with the `inactive_graph` have finished preparing, the server promotes the `inactive_graph` to `active_graph` and immediately enters the *Committing* state. This logic explains the transition from *Up* to *Committing* shown in Figure 5.2. If a worker disconnects, the server attempts to fall back to the `inactive_graph` (entering *Preparing*) or, if no valid assignment remains, reverts to the *Down* state.

5.3.3 Concurrency Model

The inference server needs to handle many concurrent requests and open WebSockets. To handle these connections, we use Tokio [tok25], an asynchronous runtime employing a thread pool for concurrency. For most communication with the server, a state change is required. Since the server's state is shared, this would require exclusive locking for most requests, leading to lock contention, increased complexity, and potential deadlocks. Instead, we employ a hybrid threading model: web connections run asynchronously using Tokio's threadpool, and the inference server uses a single-threaded synchronous event loop. The two parts communicate using non-blocking events with channels.

This threading model allows for hundreds of concurrent web requests while keeping the inference server's event loop single-threaded and synchronous. We avoid locking and parallel state updates, simplifying the event loop and allowing us to focus on the inference logic rather than concurrency. Since most messages result in very little computation, a few HashMap lookups at most, we are unlikely to bottleneck at the server unless we scale to thousands of concurrent inference requests and workers. Since scaling and throughput are not part of our design goals (see 4.1), there are no real disadvantages to using this concurrency model.

Evaluation

In this chapter, we evaluate the inference server using a dedicated test bed. We will run the distributed LLM inference with different configurations. We start by detailing the test bed, models, and metrics. Next, we establish the baseline measurements. Then we will perform experiments tailored to our research questions. Lastly, we provide a short overview of our findings.

6.1 Experimental Setup

6.1.1 Testbed

Device	CPU	RAM	GPU
Littlecorn	AMD Ryzen 7 5700G	64 GB DDR4-3200	RTX 4070 Ti (12GB)
Aidos	AMD EPYC 7452	96 GB (6x16GB)	4x RTX A4000 (16GB)
Lenovo ThinkStation PGX 30KL0004GF	ARM Cortex-X925	128 GB LPDDR5x	GB10 Grace Blackwell
Jetson AGX Thor	ARM Neoverse-V3AE	128 GB LPDDR5x	Blackwell (2560-core)
Jetson AGX Orin	ARM Cortex-A78AE	64 GB LPDDR5	Ampere (2048-core)
Jetson Orin Nano	ARM Cortex-A78AE	8 GB LPDDR5	Ampere (1024-core)

Table 6.1: Device Hardware Specifications

#Workers	Devices Used	CPU/GPU Used
1	Aidos (GPU 0)	RTX A4000
2	Aidos (2x)	2x RTX A4000
3	Aidos (3x)	3x RTX A4000
6	Aidos (4x) + ThinkStation + Thor	4x RTX A4000 + CPUs
10	Aidos (4x) + ThinkStation + Jetson (5x)	4x RTX A4000 + CPUs

Table 6.2: Worker assignments for each experiment configuration

Hardware For evaluation, we use a maximum of 10 workers, with each worker assigned to a dedicated CPU or GPU. Table 6.1 provides a summary of the devices used. For all experiments, the distributed inference server is run on Littlecorn. This device is not used as a worker during experiments. Aidos has 4 GPUs; when running experiments with 1 or 3 workers, these workers are run on Aidos. Only during experiments using 10 workers are the other devices used as well. Together with Lenovo ThinkStation PGX, Jetson AGX Thor, Jetson AGX Orin, and three Jetson Orin Nano, we have a total of 10 workers. Table 6.2 provides an overview of the worker assignments for different configurations.

Software Environment Regarding software, our code includes lock files and fixed compiler versions. We used Rust *nightly-2026-01-15* to compile the server. Littlecorn runs Ubuntu 22.04.4 LTS, Aidos runs Ubuntu 24.04.1 LTS, ThinkStation runs Ubuntu 24.04.3 LTS, Thor runs Ubuntu 24.04.3 LTS, and all other Jetson devices run Ubuntu 22.04.5 LTS. Python workers use Python 3.12, ONNX Runtime 1.23.2, and CUDA 12.

Table 6.3: Worker Network Statistics

Worker Type	Bandwidth (MB/s)	Latency (ms)	TPS
aidos	117.63	0.54	57.79
AGX Thor	31.77	1.20	48.23
ThinkStation	11.76	1.61	25.21
AGX Orin	29.50	1.97	19.30
Orin Nano	24.71	2.19	10.53

Network All devices in the test bed are connected via LAN and are in close physical proximity. The latency between devices ranges from 500 microseconds to 2.5 milliseconds. The bandwidth ranges from 10 to 120 MB/s. These measurements were obtained by the server while running experiments. The workers download random data from the server for 5 seconds and report their bandwidth. Regarding latency, we use periodic heartbeats and take the median. Table 6.3 shows these metrics for each worker.

6.1.2 Models

For evaluation, we use three models: Qwen/Qwen3-0.6B, Qwen/Qwen3-1.7B, and Qwen/Qwen3-4B. Using Microsoft Olive, we convert the models to ONNX using the command:

```
olive optimize --model_name_or_path Qwen/Qwen3-0.6B \
  --device gpu --provider CUDAExecutionProvider \
  --output_path ../../../../models/Qwen3-0.6B-olive/
```

Model	Parameters	Size	Layers	hidden_size
Qwen/Qwen3-0.6B	0.6B	2.4 GB	28	1024
Qwen/Qwen3-1.7B	1.7B	6.9 GB	28	2048
Qwen/Qwen3-4B	4B	16 GB	36	2560

Table 6.4: Characteristics of ONNX models. Models were converted using Microsoft Olive.

Table 6.4 shows the model characteristics for each converted model. Since we are using full-precision models, we observe that model size grows significantly with increasing parameter size. Although Qwen3:0.6B fits in memory on most devices, Qwen3:4B already requires splitting the model for most consumer graphics cards. The number of transformer layers also increases for the 4B variant. For networking, the increase in `hidden_size` becomes relevant. Larger models require transferring larger tensors over the network.

6.1.3 Metrics

In our experiments, we use 5 metrics. The main metric to evaluate the inference server is tokens per second (TPS). TPS is arguably the most widely used metric for evaluating LLM inference. It indicates how many tokens are generated per second during inference. This includes all generated tokens; the prompt is not included in the measurement. Time to first token (TTFT) and time per output token (TPOT) are more fine-grained measurements. As described in 2.1.3, there can be a large difference between generating the first token and subsequent tokens during autoregressive generation. TTFT measures the time it takes to generate the first token given the input prompt. TPOT measures the average time taken to compute each token, excluding the first one.

In addition to these LLM-specific metrics, we use transferred bytes and time. During runtime, the server accumulates the amount of bytes transferred over the network for all WebSocket communication with the workers. For time, we use timestamps in the server logs.

During runtime, the server logs information in 3 ways: Log level *INFO* is logged to stdout, log level *DEBUG* is logged to a file, and metrics are collected at level *TRACE* and logged as NDJSON to a file. This allows for easy parsing of metrics. The metrics TPS, TTFT, and TPOT are logged by the server at the end of each request. To obtain the transferred bytes, we take the difference between the total at the end of an inference request and the total at the start. The time per request is parsed the same way.

6.1.4 Limitations

Our evaluation has the following limitations:

WebGPU Constraints We were unable to run WebGPU on Jetson devices, limiting our web worker experiments to x86 machines. Additionally, Puppeteer (headless Chrome) does not support FP16, so we must use FP32 for all experiments. Native browser workers support FP16, so our web worker measurements represent worst-case performance.

We therefore include web workers for our baseline measurements with 1 worker. For other evaluations, we use native workers.

Quantization We were unable to produce working quantized ONNX models using the available tools (Olive, Optimum). All experiments use FP32 precision. Production deployments would use INT4 or FP16 quantization for better performance.

We tried the following with respect to quantized ONNX models:

- **Use models on Huggingface:** There are quantized ONNX models such as `onnx-community/Qwen3-0.6B-ONNX`. The 0.6B and 1.7B parameter versions worked; however, the 4B variant did not produce coherent outputs.
- **Quantize using tools:** We tried quantizing LLMs using Python and Microsoft Olive. Olive worked on Qwen3:0.6B, producing a working INT4-quantized model. The 1.7B and 4B quantized models did not produce coherent outputs. Other tools did not work either (Optimum, `onnxruntime_genai.models.builder`).

Since quantized models are not well supported by WebGPU, we decided to run experiments using full-precision.

Jetson and ONNX ONNX Runtime with GPU is not well supported on Jetson. While there are some pre-built python wheels and other tooling, we were unable to implement a python worker for Jetson using ONNX Runtime with CUDA. We tried implementing a custom worker for Jetson by converting the ONNX model using TensorRT; however, the model produced by Olive is not compatible with TensorRT. Due to time constraints, we decided not to investigate further and used ONNX Runtime using the CPU on Jetson devices.

Model Diversity Due to time constraints, we only evaluated the Qwen3 family. We were able to run other models, such as Phi-4, Gemma 2, and gpt-oss-20B; however, they are not included in our evaluation.

6.2 Baseline

6.2.1 Experimental Design

Objective

Establish a strong baseline that shows insights across configurations and key metrics. This baseline is used in combination with more specific measurements in later chapters

to answer our research questions.

Methodology

Unless otherwise specified, all baseline experiments are run 10 times per configuration. We use the mean in our results. To reduce variance, we run experiments sequentially with one active inference request at a time, we use greedy decoding resulting in deterministic output, and we use the same prompt across all runs.

We use 3 prompting strategies:

- **Chat:** We use a short prompt to generate a short answer. The prompt in this case is 'Why is the sky blue?', resulting in an input sequence length of 25 when including the prompt template. We run token generation until 128 tokens are generated.
- **Creative:** We use a short prompt to generate a long response. The response in this case is 1024 tokens long. We use the prompt 'Write a 10 page novel about the blue sky?.'
- **Summarization:** We use a long prompt to generate a short response. Again, we generate 128 tokens.

Since we use greedy decoding in the distributed inference server, we get deterministic runs. Additionally, we set a fixed generation length, so we generate until the specified sequence length is reached even if the EOS token was encountered.

We compare our distributed inference system against:

- **Reference: Ollama:** For context, we include measurements from Ollama running on Aidos. Ollama uses 4-bit quantization by default, so this is not a direct comparison since our models use full-precision. However, Ollama represents the state-of-the-art in consumer LLM inference and provides a reference point for what users might expect. To get a reproducible result, we use the Ollama API and set a seed along with a maximum number of tokens to generate:

```
curl http://localhost:11434/api/generate -d '{
  "model": "Qwen3:0.6B",
  "prompt": "Why is the sky blue?",
  "stream": false,
  "options": {
    "num_predict": 128,
    "seed": 123
  }
}'
```

- **Baseline 1: Native ONNX Runtime:** We implemented a minimal inference engine using ONNX Runtime bindings running on a single GPU. This represents ideal single-machine performance without any distribution overhead. We run this baseline on Aidos using a single RTX A4000 GPU with the same ONNX models used for distributed experiments. We evaluate three versions: A simple Python implementation, a Python version using `ORTModelForCausalLM` from the `optimum` library, and a Rust variant using `io-binding`.
- **Baseline 2: Distributed System, Single Worker:** To isolate the overhead of our distributed architecture, we run our distributed inference server on Littlecorn with a single worker on (Aidos). This measures the overhead of WebSocket communication, serialization, and coordination logic with minimal network latency.
- **Baseline 3: Distributed System, Single Worker (Web):** Similar to Baseline 2, but using a web worker (Puppeteer in headless Chrome) instead of a native Python worker. This quantifies the overhead of browser-based execution (WebGPU vs CUDA).

Expected Results

We expect that

- Larger models have strictly worse performance across all specified metrics compared to smaller ones.
- Longer prompts lead to worse performance, as required computation increases with sequence length.
- Ollama beats other engines by a large margin because of its use of quantized models and optimized kernels. In terms of TPS, web-based inference is expected to achieve 70-80% of native performance.

6.2.2 Results

Figure 6.1 shows tokens per second across engines and prompt strategies. We observe that Ollama achieves the best performance across all strategies. The optimum ORT engine achieves the worst performance, all other native clients have roughly the same TPS. As mentioned in 6.2.1, Ollama is only included as a reference. Ollama uses quantization and is highly optimized for LLM inference; as a general purpose inference engine, ONNX Runtime is bound to have worse performance, especially when using full-precision models.

Regarding the prompt strategy, longer prompts lead to slower TPS. Since *chat* has the smallest total sequence length, its TPS are the lowest. The strategies *creative* and *summary* end up with roughly the same token count; however, *creative* has a lower TPS because we average values starting with shorter sequences. Using ONNX Runtime results in much greater performance degradation with increasing sequence length than Ollama.

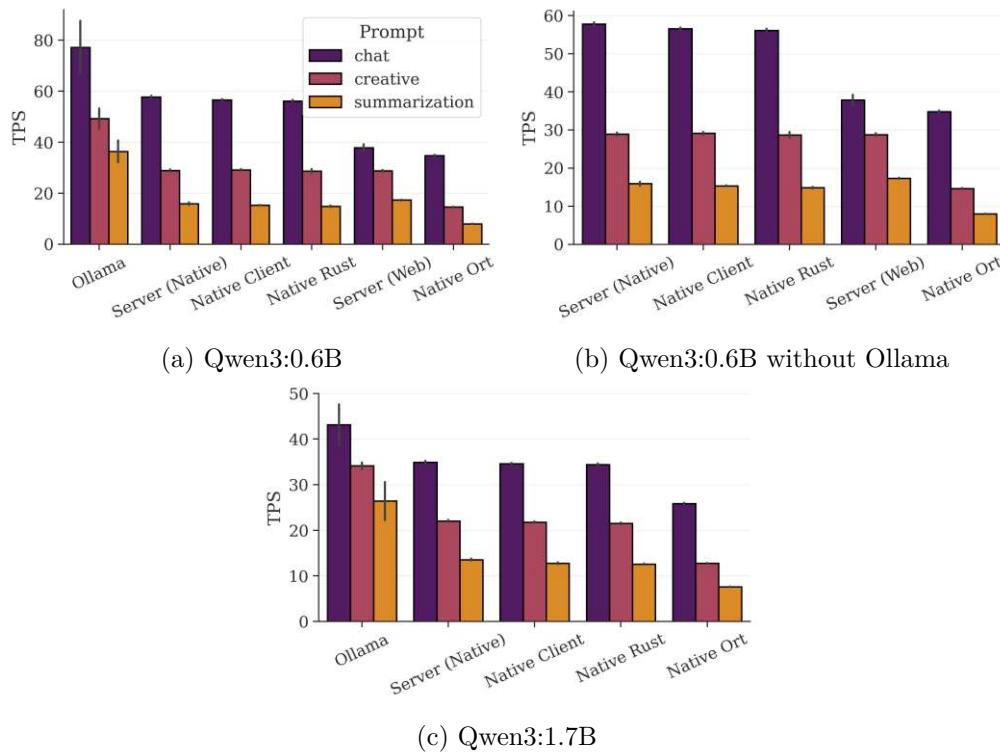


Figure 6.1: Comparison of inference engines in terms of TPS for different prompting strategies.

This can, in part, be attributed to the fact that ONNX Runtime does not implement KV cache-specific optimizations. For each forward pass, the entire cache needs to be reallocated in GPU memory. LLM inference engines such as Ollama use optimized kernels and attention mechanisms to speed up inference.

For Qwen3:0.6B we additionally include metrics for a web worker. For *chat* the TPS are lower than for native inference. Interestingly, for longer prompts, the web worker achieves roughly the same performance as native workers. It is unclear why this occurs; one possible explanation is that ONNX Runtime Web implements kernels differently for WebGPU, which leads to this improvement.

In Figure 6.2, we compare different workers running inference for the *chat* prompt using Qwen3:0.6B. We observe that Aidos achieves the 58 TPS, the highest value. This is expected since it is the only worker using the GPU. Even though Thor runs on the CPU, it achieves 48 TPS, beating Aidos' WebGPU implementation. Using Aidos as a web worker instead of a native worker results in a performance decrease of approximately 33%. Nano achieves the worst performance with only slightly more than 10 TPS.

As shown in Figure 6.3, both increasing model size and the number of workers leads to strictly worse performance across all metrics. Regarding TPS, TTFT, and TPOT, using

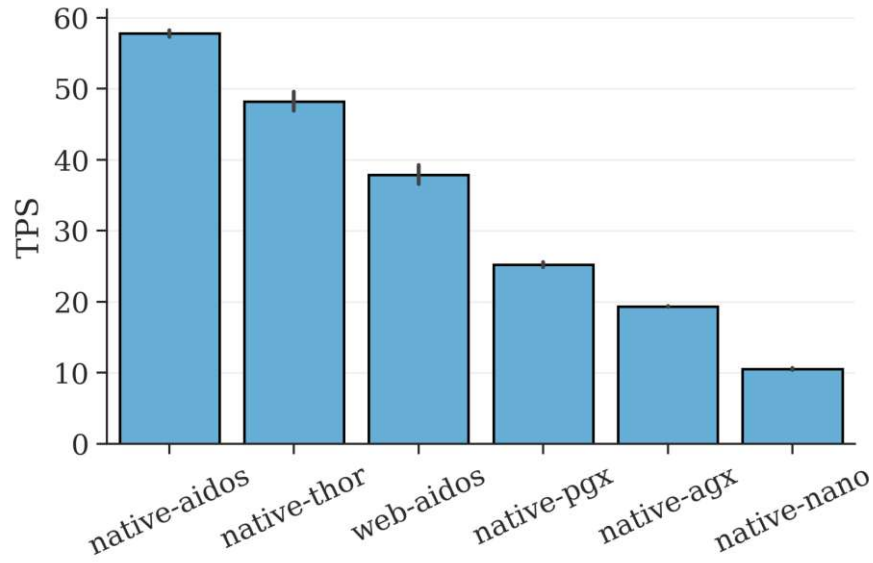


Figure 6.2: Comparison of workers using the *chat* prompt with Qwen3:0.6B.

6 or 10 workers leads to a notable decrease in performance compared to using fewer than 4 workers. Especially when using all 10 workers, performance drops significantly. This is expected since we use Thor and ThinkStation for 6 workers in addition to Aidos. For 10 workers, we use all workers. This is expected since Nano is the slowest worker (see 6.2). Regarding the number of bytes transferred per complete inference request, we see that it scales linearly with the number of workers. When using a single worker, the transferred bytes are only a few KB because no intermediate tensors need to be transferred. The input of the model is generally small, and the output is optimized to return only the next token (see 5.2.2).

6.3 RQ2: System Overhead

6.3.1 Experimental Design

Objective

Quantify the overhead of distributed inference compared to local execution and determine how the overhead scales with the number of workers and the size of the model. Specifically, we attempt to find the overhead in terms of time not used by computation, i.e. networking and time spent on the server.

Methodology

We run inference for each model (0.6B, 1.7B, 4B) with a different number of workers (1, 2, 3, 6, and 10). For 1, 2, and 3 workers, all workers run on Aidos. For 6 workers, we

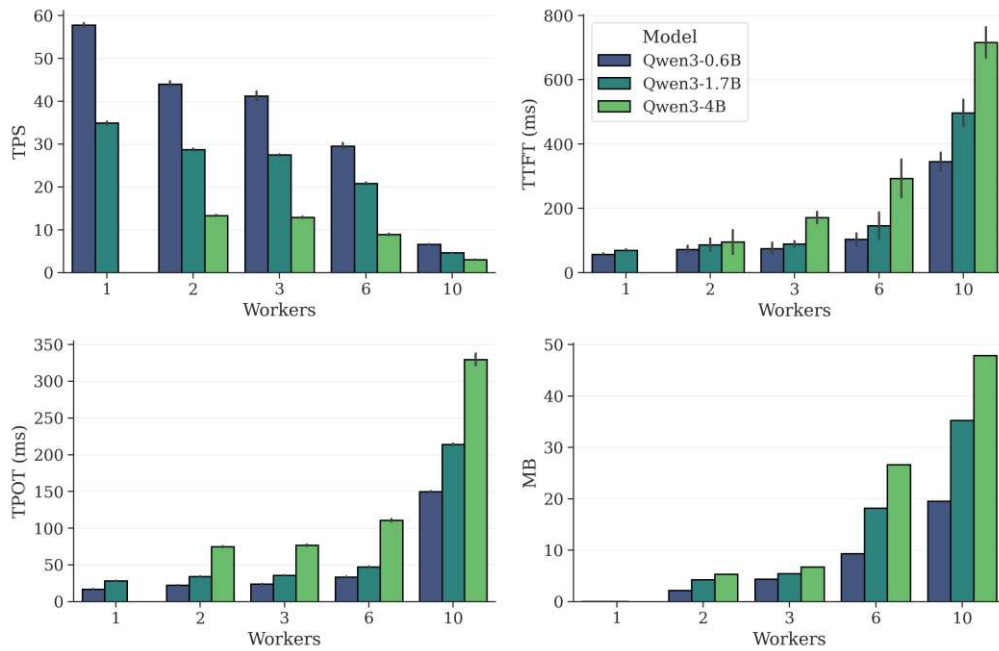


Figure 6.3: Comparison of models for across number of workers and metrics.

additionally use ThinkStation and Thor. For 10 workers, we use all the devices in the test bed (Table 6.2).

To force distribution even for small models, we limit worker memory so that all workers are used. This ensures that even for models fitting on a single worker, all workers are used and we can measure overhead.

For each configuration, we run token generation 5 times and take the mean as our value. We measure TPS and the time taken by each component of the system. The time taken is obtained from the timestamps in the logs; i.e., each request sent or received by a worker is logged with nanosecond precision. This allows us to determine exactly how much time is spent on the server. For each computation, the workers return the time it took to compute the result. The difference between the worker’s compute time and the time away from the server gives us the network time.

To establish overhead, we compare the results for different numbers of workers. We use the baseline measurements to establish the optimal inference speed.

Expected Results

We expect that

- Overhead will scale close to linearly in worker count.

- Large models will run slower than small models; however, the ratio of computation to overhead will be better.
- Network transfer time will begin to dominate as the number of workers increases.

6.3.2 Results

As shown in Figure 6.1, using the distributed inference server does not add any overhead when using 1 worker. It achieves roughly the same speed as a purely native solution. For our comparison, we will therefore use the distributed inference server with 1 worker as our baseline. Figure 6.3 shows that the inference speed decreases with the number of workers. However, when comparing the inference server with 2 and 3 workers, the performance difference is only slight. We will investigate which system component this difference can be attributed to.

As shown in Figure 6.4, the overhead of the inference server is not visible. In absolute terms, the server overhead never reaches values greater than 0.12 seconds across all configurations and inference requests. We can therefore ignore the server overhead.

The network time scales linearly with the number of workers for all models. Even in relative terms, we can see that network time starts to dominate with an increasing number of workers. Regarding the configuration with 10 workers, the computation time increases drastically. This is because we introduced Jetson Orin Nano devices that have poorer performance than other devices as shown in Figure 6.2.

Regarding the time taken by the workers, we can see that there is a slight increase in computation time from 1 worker to 2 workers, but almost no increase from 2 to 3 workers. The larger increase for 6 and 10 workers can be attributed to the heterogeneity in devices; since we do not know how the system would behave with homogeneous devices, we do not use these configurations to argue about computational overhead.

With increasing model size, network time decreases compared to computational time. We observe that for Qwen3:4B using the configuration with 10 workers, the proportion of worker computation is smaller than for other model sizes. This is an artifact in our evaluation where we were forced to assign different model proportions to the workers for different experiments. Inference requests for Qwen3:4B used proportionally more of Aidos's resources compared to the other model sizes.

6.4 RQ3: Network Optimization

6.4.1 Experimental Design

Objective

Quantify the impact of network optimization on the total number of transferred bytes and inference performance. By comparing our metrics, we can determine which optimizations fit which scenarios and how this tradeoff might evolve for different network conditions.

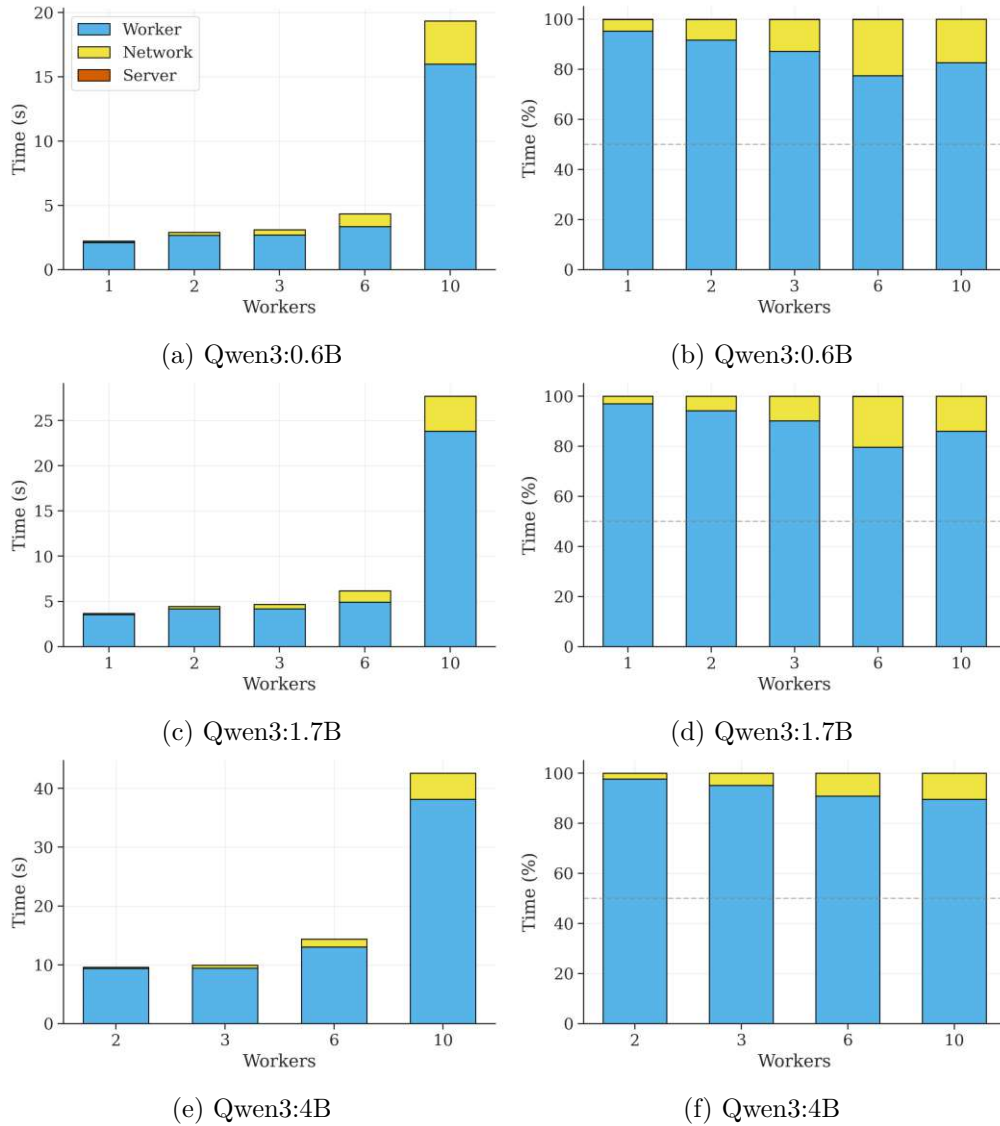


Figure 6.4: Absolute and relative time taken by each component in the system.

Methodology

As described in Section 5.2.2, we implement 5 network optimizations. In our evaluation, we will focus on 3 of them: Tensor Compression, Logit Elimination/Greedy Sampling, and Prefix KV Caching. These 3 represent tradeoffs we want to quantify; binary encoding and hash based caching are obvious choices with little downsides, so we do not evaluate them.

We design our experiment as an ablation study, turning each optimization off one by one to determine its impact on the overall system during inference. The baseline uses

all optimizations; the difference between them gives an indication of each optimization’s importance.

We run our experiments 5 times for each configuration taking the mean. We evaluate the inference server for different model sizes (0.6B, 1.7B, and 4B) and different numbers of workers (1, 3, and 10). For each configuration, we measure the TPS, TTFT, TPOT, and bytes transferred averaged per request.

Expected Results

We expect that

- Prefix KV caching will have little impact on TPS and TPOT; however, it will decrease TTFT. Although this decrease will also be marginal since our input prompt is short.
- Logit elimination will have the greatest impact, reducing bytes transferred and inference speed.
- Compression using zstd will decrease the amount of bytes transferred by roughly 30%. Although compression works well for structured data, Protocol Buffers are already optimized, and it remains to be seen whether tensors benefit from compression. Overall, compression is expected to lead to a performance increase across all metrics, although this is offset slightly, since compression incurs an additional computational cost.
- The baseline will have the lowest number of transferred bytes as well as the best inference performance.

6.4.2 Results

Table 6.5: Network Stats (bytes) for Qwen3-0.6B

Optimization	1 Worker	3 Workers	10 Workers
Baseline	0.02	4.36	19.52
No Compression	0.33	4.78	21.06
No Greedy	75.57	79.90	95.06
No Prefix Cache	0.03	4.78	21.43

Tables 6.5, 6.6, and 6.7, show that the baseline with all optimizations enabled has the lowest number of transferred bytes in all configurations. Logit elimination achieves the greatest reduction in transferred bytes of roughly 75 MB per request. We can verify this amount: Our *chat* prompt template has a total sequence length of 153 tokens. Logits have a shape `[batch_size, sequence_length, vocab_size]`, resulting in an overall size of `[1, 153, 151936]` and roughly 23.2 million elements to be transferred.

Table 6.6: Network Stats (bytes) for Qwen3-1.7B

Optimization	1 Worker	3 Workers	10 Workers
Baseline	0.02	5.44	35.25
No Compression	0.11	7.02	39.11
No Greedy	75.07	80.49	110.31
No Prefix Cache	0.02	5.93	38.74

Table 6.7: Network Stats (bytes) for Qwen3-4B

Optimization	3 Workers	10 Workers
Baseline	6.77	47.88
No Compression	8.72	51.59
No Greedy	85.65	122.86
No Prefix Cache	7.38	52.67

At a size of 4 bytes per element, we get a total of 92 MB. When moving token sampling inside our ONNX graph, we instead need to send a single 64-bit integer back to the server. Since compression and Prefix KV caching are still enabled, we can see that 75 MB is within expectations.

Regarding zstandard tensor compression, we observe a compression ratio between 10% and 30% in transferred bytes. When using only a single worker, this ratio is much higher; however, this is due to the fact that the attention mask that needs to be transferred has a shape of `[batch_size, total_sequence_length]` consisting of only the number 1. This entire tensor is compressed by zstd leading to a much higher ratio. We can conclude that intermediate tensors consisting of 32-bit floating point numbers do not benefit much from compression.

Our *chat* prompt used for testing has a static prompt token length of 25 tokens, 14 of which can be cached using prefix KV caching. This means that we reduce the shape of intermediate tensors that are transferred when generating the first token from `[batch_size, 25, hidden_size]` to `[batch_size, 14, hidden_size]`. This is in line with our results of a few dozen KB, which increase with `hidden_size` and the number of workers.

Figure 6.5 shows TTFT and TPOT in model sizes and network optimizations. Lower values indicate better performance. We observe that the models exhibit the same characteristics for all configurations.

Using zstd for tensor compression results in worse performance for most configurations. The computational overhead compression is larger than the benefit of size reduction. In our evaluation, the network has fast interconnects; for slower networks, this trade off is expected to shift.

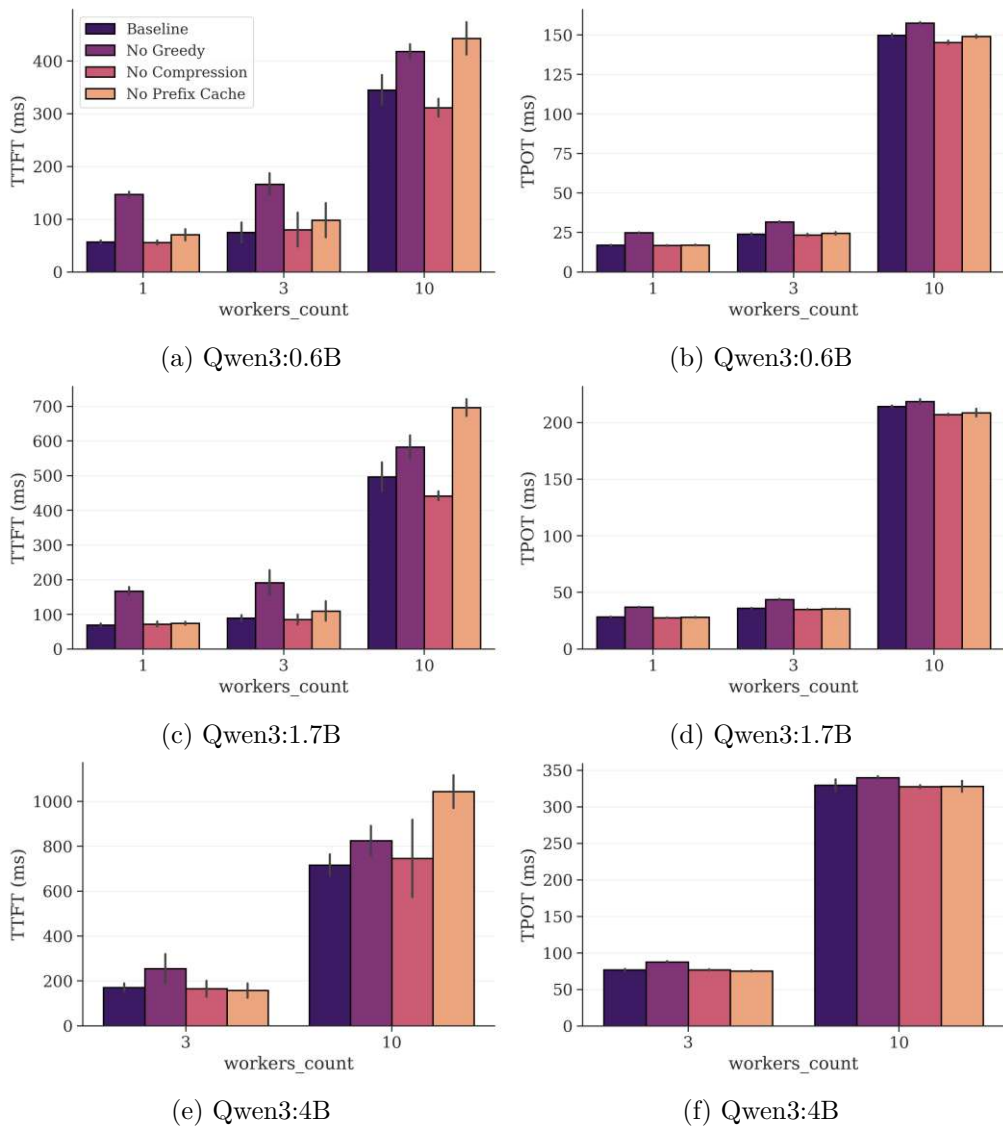


Figure 6.5: Comparison of optimization strategies in terms of TTFT and TPOT. For the baseline all optimizations are enabled. Otherwise a single optimization is disabled at a time.

Using greedy decoding inside ONNX results in the biggest improvement, both in terms of TTFT and TPOT. The optimization adds almost no overhead, but decreases the amount of bytes transferred by a large amount. The only tradeoff is that the server does not have full control over token generation. However, it should be possible to modify the ONNX graph to allow for other methods of token generation, removing this tradeoff and yielding only benefits at the cost of a slight increase in complexity.

Prefix KV caching has no impact on TPOT. This is expected as the generation of

subsequent tokens remains the same with or without the optimization as long as no worker disconnects. Regarding TTFT, prefix caching results in a speed increase of about 30%. Our prefix cache is only 14 tokens long; performance is expected to increase further with longer system prompts.

6.5 RQ4: Fault Tolerance

6.5.1 Experimental Design

Objective

Determine the robustness of the distributed inference server to unexpected worker disconnects. We determine bottlenecks for different model sizes and prompting strategies by observing the server in extreme scenarios with highly volatile workers.

Methodology

We start our inference server as normal with all optimizations enabled. Workers are started using a wrapper script that gradually increases churn rate. For each configuration, we run the experiment over 600 seconds, increasing the churn rate from 0 up to a target mean time to failure of 10 seconds, i.e. on average the worker fails once every 10 seconds at the end. After disconnecting, the worker waits between 1 and 10 seconds before reconnecting.

We always connect all 10 workers, limiting the available memory of each to 6 GB. This results in the following behavior for different model sizes: Qwen3:0.6B requires 1 worker to run, Qwen3:1.7B requires 3 workers, and Qwen3:4B requires 5 workers. We use the *chat* and *creative* prompts. Before starting the experiment, we download all model weights on all workers.

As metrics, we use server TPS and server state over time. We choose server TPS over per-request TPS, as it is not guaranteed that requests will finish. Using TPS logged at 1 second intervals by the server, we get time series data showing TPS for gradually increasing churn rate. The server state can take one of four values:

- **Down:** A worker recently disconnected and the server either had no time to reschedule yet or there is no valid configuration, so the server is forced to wait for other workers to connect.
- **Preparing:** The server found a valid configuration and instructed all relevant workers to prepare their part of the model. The server is now waiting for all workers to respond.
- **Committing:** All workers finished preparing their model, i.e. they downloaded all relevant layers and are ready to activate the ONNX Runtime session.

- **Ready:** The server is able to serve inference requests.

The server can be in two states at once; for example, the server can be ready while preparing a new assignment. In these cases, we set the server state as the stronger of the two, i.e. as ready in our example.

Expected Results

We expect that

- The server’s generation rate will decrease as the churn rate increases.
- Larger models are much more affected by the increasing churn rate.
- For smaller models, the server spends most of its time serving requests or committing. For large models, more time is spent preparing since the server needs to wait for 5 workers to respond, any of which can disconnect during this time.
- The *creative* prompt will perform worse than the *chat* prompt as the churn rate increases, since recomputing the KV cache for small sequence length is faster.

6.5.2 Results

As shown in Figure 6.6, the server retains high TPS for Qwen3:0.6B even with an increasing churn rate. The server spends most of its time in the ready state, even as the churn rate increases. Qwen3:1.7B starts with high TPS that drops as the churn rate increases. Qwen3:4B spends most of its time preparing and committing, only sparsely continuing with token generation.

Regarding sequence length, we observe that short sequence lengths have overall higher TPS and retain better generation rates as the churn rate increases. During the first few seconds in Figure 6.6b, we see how TPS drops as the sequence length increases. Similarly for Qwen3:1.7B the generation rate drops; however, in this case finishing the token generation takes longer due to the model’s size.

Figure 6.7 shows the proportion of time the server spends in each state as the churn rate increases. We use a rolling average of 120 seconds to obtain the proportions. As the churn rate increases, the server spends more time in non-ready states across all configurations. For larger models, which require more stable workers to be available at the same time, the non-ready states quickly start to dominate. For Qwen3:0.6B, almost no time is spent in the *Down* or *Preparing* states. Since there are 10 workers and only 1 is required to serve the model, almost all of the time at least 1 worker configuration exists that is able to serve the model. Regarding *Preparing*, since all weights are already downloaded and only a single worker needs to respond, the time spent in this state is essentially the latency of the worker. *Committing* takes more time, since the worker needs to create the inference session using the ONNX model. As the model size increases, more time is spent

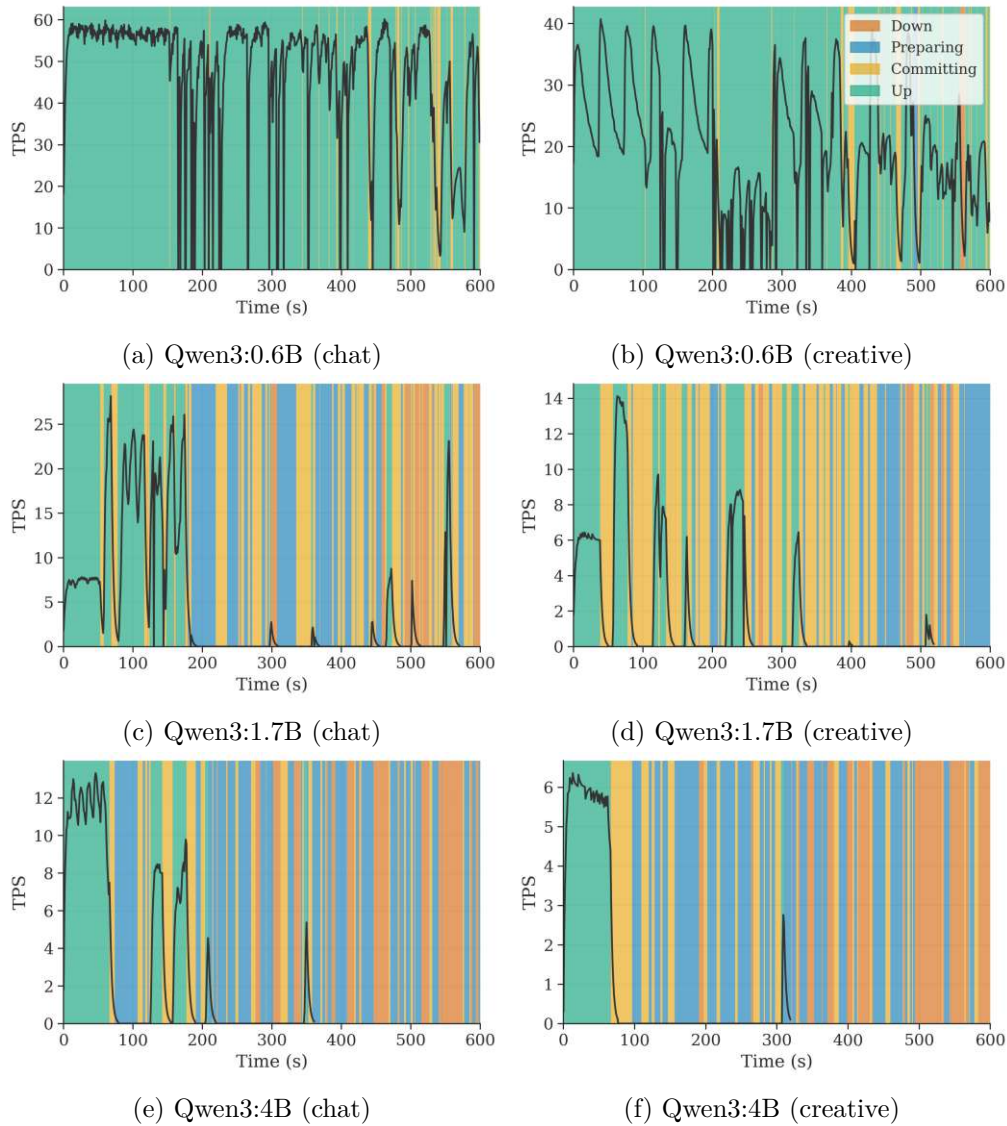


Figure 6.6: Exponentially Weighted Moving Average (EWMA) of TPS overlaid with server state changes.

in *Preparing* and *Down*. Larger models require more workers; if during preparation any worker that got assigned a part of the model disconnects, a new assignment has to be found. Similarly, if a better worker connects during preparation, the server might decide to use that worker and cancel the previous assignment.

As shown in Table 6.8, the time spent in each state shifts from the *Up* state towards the *Down* state as the size of the model increases. Table 6.9 shows that the number of worker disconnects is roughly the same for all configurations. We observe that the number of state changes is highest for Qwen3:1.7B, which requires 3 workers to run. This is because

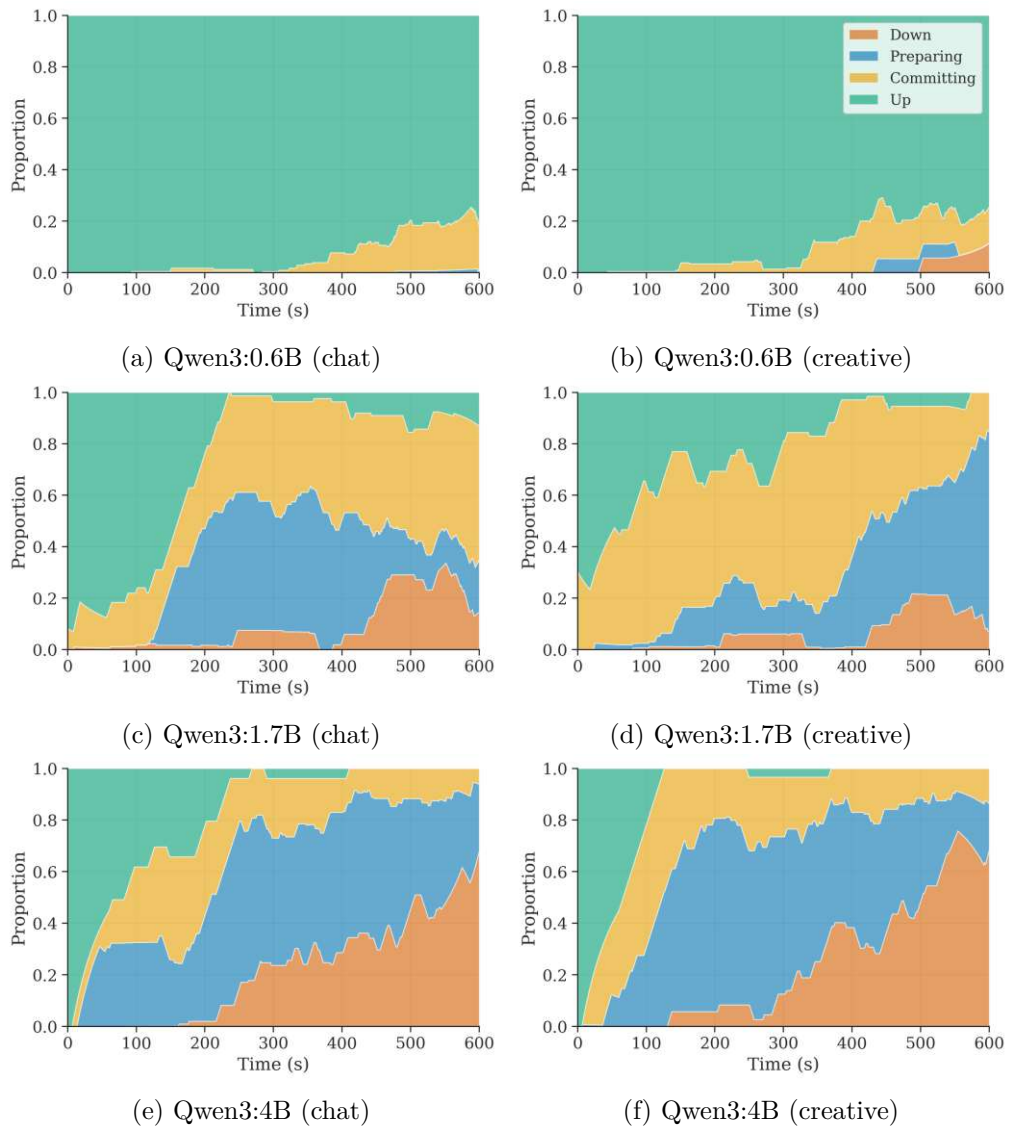


Figure 6.7: The proportion of time the server spends in each state aggregated using a rolling window of 120 seconds.

for Qwen3:0.6B a worker disconnect rarely results in a state change since only 1 worker is used. For Qwen3:4B the number is lower since we are using more workers, so each state change in *Preparing* and *Committing* takes longer to complete.

Tables 6.10, 6.11, and 6.12 show the state transitions in more detail. For Qwen3:0.6B, almost every transition is a successful transition towards state *Up*. Every time the server is repartitioning this is successful and leads to *Committing*, from where the server transitions to *Up* for most cases. In contrast, for Qwen3:4B the server spends most of its time trying to assign new partitions to workers. Specifically, the transition *Preparing*

Table 6.8: Fault Tolerance Statistics

Model	Prompt	Down	Preparing	Committing	Up	Availability (%)
Qwen3-0.6B	chat	0.77	1.18	34.81	563.23	93.87
Qwen3-0.6B	creative	7.12	6.69	46.18	540.01	90.00
Qwen3-1.7B	chat	55.69	170.87	204.45	169.00	28.17
Qwen3-1.7B	creative	35.63	146.52	277.28	140.57	23.43
Qwen3-4B	chat	126.91	250.05	111.14	111.91	18.65
Qwen3-4B	creative	141.13	258.82	130.08	69.97	11.66

Table 6.9: Fault Tolerance Statistics

Model	Prompt	Down	Preparing	Committing	Up	Disconnects
Qwen3-0.6B	chat	27	27	43	33	215
Qwen3-0.6B	creative	29	29	47	37	228
Qwen3-1.7B	chat	46	94	81	23	237
Qwen3-1.7B	creative	45	80	90	25	229
Qwen3-4B	chat	34	125	39	7	198
Qwen3-4B	creative	30	124	40	3	224

Table 6.10: State Transitions - Qwen3-0.6B, chat

From	To	Time (s)	Count	Mean (s)
Down	Preparing	0.77	27	0.03
Preparing	Committing	1.18	27	0.04
Committing	Down	0.32	1	0.32
Committing	Committing	8.06	9	0.90
Committing	Up	25.59	32	0.80
Up	Down	563.22	26	21.66
Up	Committing	0.01	7	0.00

to *Preparing* shows that the server is constantly trying to repartition towards better assignments. This occurs when a better worker connects and the server attempts to use it.

Looking at the mean time spent in each transition, we observe that during transitions towards *Up*, the server spends less than a second on average in *Preparing*. Most of the time is spent in the *Committing* state.

Table 6.11: State Transitions - Qwen3-1.7B, chat

From	To	Time (s)	Count	Mean (s)
Down	Preparing	46.89	45	1.04
Preparing	Down	4.32	1	4.32
Preparing	Preparing	117.36	40	2.93
Preparing	Committing	49.19	53	0.93
Committing	Down	52.51	33	1.59
Committing	Preparing	27.34	9	3.04
Committing	Committing	54.05	17	3.18
Committing	Up	70.55	22	3.21
Up	Down	168.94	12	14.08
Up	Committing	0.06	11	0.01

Table 6.12: State Transitions - Qwen3-4B, chat

From	To	Time (s)	Count	Mean (s)
Down	Preparing	126.91	34	3.73
Preparing	Down	43.19	8	5.40
Preparing	Preparing	184.91	80	2.31
Preparing	Committing	21.93	36	0.61
Committing	Down	41.54	22	1.89
Committing	Preparing	41.65	10	4.17
Committing	Committing	2.47	1	2.47
Committing	Up	25.48	6	4.25
Up	Down	107.29	4	26.82
Up	Preparing	4.61	1	4.61

6.6 Summary of Findings

This evaluation establishes the viability and limitations of web-based distributed LLM inference across three dimensions: system overhead, network optimization, and fault tolerance.

Distributed Inference Overhead is Minimal on Homogeneous Hardware Our baseline measurements (Section 6.2) demonstrate that the distributed system with a single worker achieves performance parity with native ONNX Runtime, confirming negligible coordination overhead. When scaling to multiple workers on homogeneous hardware (3 GPUs on Aidos), network time remains below 25% of total execution time for all models (Section 6.3). The server orchestration overhead never exceeds 120ms per request, representing less than 0.2% of the inference time across all requests. However, worker heterogeneity introduces significant variance: Jetson Nano devices using CPU are more

than 5x slower than RTX GPUs, causing compute time instead of network time to dominate in mixed configurations.

Network Optimizations Provide Asymmetric Benefits Ablation studies (Section 5.2.2) reveal that network optimizations have dramatically different cost-benefit profiles. Logit elimination provides the largest impact, reducing bytes transferred by 75MB per request (a 5x reduction for 10 workers; a 10x reduction for 3 workers) with negligible computational overhead. This optimization alone makes distributed inference viable for slow networks. Prefix KV caching reduces TTFT by 30% for our 14-token system prompt; the benefits scale with the length of the system prompt. Tensor compression (zstd level 3) reduces the transfer volume by 10-30% but introduces computational overhead that negates the benefits on fast networks (>10 MB/s).

Fault Tolerance Depends Critically on Worker Count Chaos testing with exponentially increasing worker churn (Section 6.5) reveals a sharp viability limit based on the size of the model. Qwen3:0.6B (requiring 1 worker) maintains more than 70% uptime even at with a mean-time-to-failure of 10 seconds; overall spending over 90% of time in the *Ready* state. Qwen3:1.7B (3 workers) shows graceful degradation, maintaining 20-30% uptime at moderate churn rates. In contrast, Qwen3:4B (5 workers) spends less than 20% of the time in the *Ready* state even with low churn. For higher churn rates, the server is unable to reliably serve inference requests. State transition analysis reveals that for higher worker counts, most of the time is spent attempting to assign model partitions. *Preparing* takes a mean of about 2 seconds for 5 workers, during which any worker disconnect or reconnect can trigger repartitioning. This suggests that with increasing model size participants need to be more reliable to guarantee stable inference.

Discussion

In this chapter, we highlight our key findings and results. We investigate design tradeoffs and provide alternatives. We position this work in the broader field of AI and offer a guide for real world deployment. Lastly, we go over the limitations of the proposed architecture and offer directions for future work.

7.1 Interpretation of Results

In this section, we will start by answering each research question individually. We will then provide insight into the practicality of web-based distributed inference and how it compares with other inference solutions.

7.1.1 Research Questions

RQ1: Zero-Setup Distribution via Web Technologies

How can web technologies enable zero-setup distributed LLM inference across heterogeneous consumer hardware?

Our goal is to run a server that hosts a website where users can contribute their computational resources towards serving a model. By building such a website and the corresponding server, we can achieve zero-setup distributed inference. Since the goal is to run in the browser, all our technologies need to be tailored towards this.

We started by investigating different engines to run machine learning models in the browser, such as WebLLM [RQZ⁺24] that uses TVM [CMJ⁺18] to compile to WebGPU or transformers.js [Tra], a port of the transformers Python library for the web. To run non-standard models, which we need for distributed inference, the inference engines need

to support essentially arbitrary machine learning models. We concluded that ONNX Runtime Web is a great choice; this decision is supported by transformers.js, which also uses it. ONNX Runtime can run most machine learning models as long as all operators are supported, and the JavaScript library is easy to use and build upon. To run machine learning models in the browser using the GPU, ONNX Runtime Web supports WebGPU. As shown in Section 6.2, the use of WebGPU incurs some overhead compared to native execution.

Regarding the communication protocol, we chose WebSockets. WebSockets support binary communication, add almost no overhead over raw TCP, support communication, and are widely supported. These benefits in addition to browser compatibility requirements make WebSockets a great choice. In Section 4.4 we explain this choice in more detail.

To facilitate binary communication, we encode our messages as Protocol Buffers. We briefly investigated other options such as FlatBuffers; however, since there is little difference between the two for our use case and ONNX uses Protobuf for encoding as well, we chose Protobuf due to its smaller footprint in terms of required dependencies.

Regarding network topology and system architecture, we chose a centralized approach. A purely decentralized version is not possible since a server is always required to serve the website; therefore, we went the opposite direction, using a purely centralized server. This allows for easier coordination of workers and collection of metrics.

RQ2: Overhead of Distributed Inference

What is the overhead of distributed LLM inference compared to local native execution?

The maximum overhead the inference server added across all inference requests was less than 0.15% of the total inference time. This is negligible; we will therefore ignore the server overhead in the rest of this discussion and only consider the network and computational time.

As shown in Section 6.2, the distributed inference server adds no overhead over purely native solutions for 1 worker. In fact, it performs slightly better than the baseline ONNX Runtime implementation due to optimizations such as token decoding inside ONNX and prefix KV caching.

Regarding the 2 and 3 worker configurations running on the same hardware, using 2 workers instead of 1 incurs a performance drop of 23.8% and 17.8% for Qwen3:0.6B and Qwen3:1.7B, respectively. When using 3 workers instead of 2 the jump is 6.4%, 4.3%, and 3.1% for Qwen3:0.6B, 1.7B, and 4B respectively. We can see that performance drop-off flattens out, the computational overhead of using more workers only increases slightly, and network time increases linearly. Since computational time dominates network time in our test bed with low latency, we can conclude that the distributed inference server scales well to roughly 10 workers before network time starts to dominate for the models

we tested. For even larger models, computational time would take up a larger percentage, so even more workers could be added before network time starts to dominate.

Regarding inference in the browser, using WebGPU decreases TPS by about 33% for short sequence lengths. Surprisingly, this discrepancy disappears for longer sequence lengths. It is unclear why WebGPU scales better with longer sequence lengths; we have 2 possible explanations, both of which might contribute: Launching GPU kernels in browsers is generally more expensive since the correctness of shaders has to be verified more rigorously to prevent malicious actors from accessing GPU memory (see Section 2.4.2). The overhead becomes less pronounced as the sequence length, tensors, and computation time increase. Secondly, it is possible that ONNX Runtime Web optimizes kernels differently for WebGPU than for CUDA, which may explain this improvement.

RQ3: Network Optimization Trade-offs

To what extent do specific transport optimizations reduce transfer volume and improve inference speed in bandwidth-constrained environments?

We implemented 3 transport optimizations aimed specifically at reducing transfer volume and improving inference speed:

- **Logit Eliminations:** By moving token decoding into the ONNX model graph, we can remove the transfer of logits over the network. Instead, we just need to transfer the next token.
- **Compression:** We use zstd compression on large tensors to reduce the transfer volume.
- **Prefix KV Caching:** We compute the KV cache for the system prompt at server startup and transmit it to new workers. This allows us to reduce the transfer volume for the generation of the first token.

As shown in Section 6.4, logit elimination had the greatest impact on transfer volume and generation speed. We reduced the amount of bytes transferred by about 75 MB per inference request for 128 generated tokens. TPOT was reduced by 11.2%.

Compression leads to a decrease in transfer volume between 10% and 30% depending on the size of the model and the number of workers; we expected an average compression ratio of roughly 30%. This is likely because intermediate tensors are entirely 32-bit floating point numbers with little structure to be captured by zstd. This resulted in the computational overhead of compression exceeding the performance benefit of transferring less data for our test bed. Instead of using zstd as the compression algorithm, floating-point-specific compression algorithms might have yielded better results.

Prefix KV Caching reduces transfer volume by a few KB up to 4.8 MB depending on the model size and worker count. This improvement is only relevant for the generation of the first token, so we see no speed improvement in terms of TPOT. However, TTFT decreases by up to 25% for 10 worker configurations.

Overall, our optimizations have limited impact on transfer volume and decoding time. This is because there is little room for improvement in the first place. The biggest optimizations are splitting the model at layer boundaries to produce smaller intermediate tensors, not syncing the KV cache during generation, and using a binary format instead of JSON. Applying logit elimination, we enter a scenario in which the transferred tensor does not exceed a few KB.

RQ4: Fault Tolerance Under Worker Churn

How does system throughput degrade as worker churn rate increases?

In Section 6.5, we show how the distributed inference server deals with unexpected worker disconnects and a highly volatile environment. We observe that the system remains stable and maintains high generation speeds, even with only a small number of workers required to load the model. In a 10-worker configuration where 1 worker is required to load the model, the system recovers in less than a second and continues serving requests. In this configuration, we achieve availability of more than 90% while gradually increasing the churn rate from a mean time to failure of 0 to 10 seconds over 10 minutes.

When more workers are required to load the model, the system cannot reliably serve requests. With an average disconnect rate of once every 3 seconds, a model that requires 5 workers cannot repartition, distribute, and continue generation fast enough. To achieve high TPS for stable scenarios, the system is designed to evaluate new workers and possibly repartition in response to a new worker connecting. In our evaluation, this leads to a scenario where the server is constantly attempting to assign better partitions, only for a worker to disconnect before it can continue generating. This represents a trade-off between inference speed and fault tolerance, which we will discuss in detail in Section 7.2.

7.1.2 Practical Viability

In Sections 6.2 and 6.3, we show that, given our requirement of running in the browser and our choice of ONNX Runtime, there is no overhead when running native inference compared to distributed inference with 1 worker. Adding more workers using the same hardware does not increase overhead much. We can conclude that, given our design goals and constraints, the architecture and implementation of the distributed inference server are close to optimal.

On the other hand, we observed that Ollama, a highly optimized LLM inference engine, achieves 6-7 times the inference speed. Firstly, this is because Ollama uses quantized models, where we are forced to use full-precision models due to limitations when converting

LLMs to ONNX (see 7.3.1). However, even considering this, Ollama would surpass our implementation using ONNX Runtime. ONNX Runtime is a general purpose inference engines; Ollama and similar LLM inference engines are optimized purely for LLM inference. For example, using ONNX Runtime we are forced to reallocate the entire KV cache in GPU memory for every forward pass. Using custom GPU kernels and optimizations of the attention mechanism such as FlashAttention [DFE⁺22] or PagedAttention [vll25] this is not necessary, leading to much higher generation rates and better scalability with higher sequence lengths. This shows that the current limitation of achieving high generation rates mainly has to do with limited support of the ONNX ecosystem for LLMs. With projects such as `onnxruntime-genai` [mic26], we can expect performance to improve. Alternatively, we can focus on optimizing towards native inference, implementing workers that do not use ONNX Runtime, but rather engines optimized for LLM inference. This idea is further explored in Section 7.4.1.

We conclude that our system is not a replacement for current LLM inference engines or other distributed inference solutions. Instead, it can serve as a basis for quickly testing the behavior of new LLMs that are too large to fit on a single device. For long-term deployments, we suggest investing the time to deploy more optimized and scalable engines such as vLLM [vll25] for server hardware or Petals [BRC⁺23] for distributed inference over the internet. If on-prem solutions are not required, using Cloud APIs is arguably the cheapest and best option.

Regarding the environment, the server can run on most devices as long as there is enough disk space to store the models and the network is fast enough. Regarding workers, we distinguish between web and native workers. Web workers allow for zero-cost deployment and work best on desktop devices that support WebGPU. On servers or edge devices, using native workers is preferred, as setup is required either way.

7.2 System Design and Trade-offs

Based on our requirements, we faced several design choices. In this section, we reflect on our choices based on the evaluation. We will discuss the positive and negative outcomes, offering alternatives for each trade-off.

7.2.1 Pipeline Parallelism

Early on during the design process, we had to choose between different types of parallelism. This boiled down to pipeline parallelism and operator parallelism. Due to the nature of our setting, distributed inference with low-speed interconnects over the internet, we found that using pipeline parallelism is a better fit for our goals (see Section 4.3.1). We maintain that this choice was correct except for specific edge cases. For example, an edge case is the first layer of LLM models. It contains a large tensors to convert tokens to embeddings, reaching 1.5 GB for Qwen3:4B. In cases where no device can load this tensor, for example, using multiple small edge devices, the server would be unable to run

inference even if the total amount of VRAM was enough. Another edge case is when devices have multiple GPUs. In this case larger parts of the model could be loaded on the devices and run using all GPUs with operator parallelism. This would not require an architectural change; all the worker would need to do is report the combined available memory and parse the ONNX model to be used with tensor parallelism.

7.2.2 Worker Volatility

The server was designed with arbitrary worker disconnects in mind. We showed that the system can recover from such disconnects, but it has problems with a very high churn rate. There are two main design decisions that would allow the system to run inference even with highly volatile workers. One simple change is to prevent the server from repartitioning the model unless it already has an active graph. As shown in Section 6.5, the server spends a lot of time interrupting previous assignments; this change would prevent that at the cost of taking slightly longer to reach better assignments in stable conditions. Secondly, we could switch to a distributed KV cache that is synchronized with the server (see Section 4.3.3). This would reduce bandwidth in scenarios with many interruptions. Additionally, it would allow us to use partial model assignments. Currently, inference can only run with full assignments, since each worker depends on the KV cache of the previous one. If the server had a copy of the cache, it could hot-swap individual workers without recomputing the cache. Given our assumption of arbitrary disconnects, but generally stable and long-running workers, we believe our choices to be correct. For real-world deployment with unknown participants that serve large models, we suggest revisiting this trade-off and comparing both options.

7.2.3 Star Topology

Since we are serving a website for web-based inference, a server is required. We chose a star-based topology based on this and the fact that centralized metrics collection simplifies partitioning. In our evaluation, we show that the server itself adds virtually no overhead; however, the network time scales linearly with the worker count. For lower-bandwidth networks, this becomes relevant because the server adds an extra trip for computations. Peer-to-peer solutions can directly send data from worker to worker. We believe that the centralized server as a coordinator is worth this trade-off. However, it should be possible to design the system so that the server acts only as an orchestrator, while the workers communicate only among themselves during inference. We investigate this idea further in Section 7.4.2.

7.2.4 Dynamic Repartitioning

Regarding model partitioning, we implemented a dynamic repartitioning algorithm. This choice proved essential; by minimizing end-to-end inference latency, the server automatically avoids workers with slow networks or low computational speed. Section 6.2 shows how performance can vary between workers running on powerful GPUs and workers

using low-power CPU inference. Without dynamic partitioning based on capabilities, we would be forced to either remove these slow workers or accept that they slow down inference more than necessary.

7.2.5 ONNX Runtime

Using ONNX Runtime Web worked well for deploying split LLMs in the browser. Building a prototype using ONNX Runtime Web with an already split model took only a few hours. For native inference, ONNX Runtime worked well at first; however, our evaluation showed that the inference speed is limited by the engine. In Section 7.3.1 we discuss other issues we encountered while using the ONNX ecosystem. Overall, using ONNX and ONNX Runtime worked well enough. Given that there are virtually no other options for deploying model in the web, we argue that the choice was correct. However, for this work to be usable for long term model deployments, the underlying inference engine would need to be better optimized for LLM inference.

7.3 Limitations and Threats to Validity

In this section, we explore the limitations of this work. We will focus on how these limitations affect our evaluation and on the implications for generalization to the broader area of distributed LLM inference. We will detail if and how they threaten the validity of our claims.

7.3.1 Model and Architecture

Evaluated Model Architecture In our evaluation, we used only the 3 smallest models from the Qwen3 family. Although this model family might exhibit different performance characteristics than other models, we believe this difference to be marginal. We tested other models including `microsoft/phi-4`, `onnxruntime/gpt-oss-20b-onnx`, and `google/gemma-2-2b-it`. The inference server successfully served all these models without requiring any changes beyond configuration.

Model Size Due to out-of-memory issues, we were unable to convert models larger than Qwen3:8B to ONNX. During our evaluation, we only included Qwen3:0.6B, 1.7B, and 4B. We tested Qwen3:8B; however, we did not include it in our evaluation. We believe that our system can scale to larger models such as Qwen3:32B; however, we cannot prove this conclusively. We are also not able to show performance characteristics for such models, only being able to extrapolate from the 3 models we evaluated.

ONNX ecosystem The main limitations of this thesis concern ONNX and its ecosystem. We encountered many issues at the intersection of ONNX and LLM inference that we had to either accept or work around. This starts with converting HuggingFace models to ONNX. Although some LLMs have already been converted to ONNX, such

as `onnx-community/Qwen3:0.6B`, the selection is limited. We explored different conversion utilities, including using the `onnx` and `transformers` libraries in Python, `optimum-cli`, and Microsoft Olive. All of these required a significant time investment to get started. In the end, Olive worked best for our use case. Another issue with conversion is out-of-memory problems. Using existing conversion tools, we were unable to convert models larger than the available memory, such as `Qwen3:32B`. We decided not to investigate further due to time constraints. Lastly, we were unable to run ONNX Runtime on Jetson devices. We tried using NVIDIA's Python wheels, building our application using Jetson containers, compiling ONNX Runtime from source for Jetson, and using TensorRT instead of ONNX Runtime as the inference engine. None of the solutions worked out of the box. Ultimately, we were forced to run ONNX Runtime using the CPU on Jetson devices. Although it is a limitation of our evaluation that prevents us from reasoning about overhead with higher worker counts, it does not affect the main findings of this thesis.

Quantization LLM inference is typically performed using quantized models, as full precision is generally reserved for gradient descent during training and results in significantly slower inference with only marginal gains in accuracy. Although we observed significant speedups using quantized models during development, our evaluation relies on full-precision models due to several technical hurdles. For evaluation, we used Puppeteer to run a web browser on our test-bed servers. Using `onnx-community/Qwen3:4B` quantized to `q4f16`, we found that the model runs well in desktop browsers; however, Puppeteer does not support `shader16`, which is required for 16-bit floating-point numbers. At this point, we decided not to investigate this issue further, opting to test quantized models in a desktop browser and running full-precision with Puppeteer. Next, we tried quantizing models using Olive. We were able to convert `Qwen3:0.6B` to `f16` and to `int4`. However, quantizing 1.7B and 4B using the same approach led to the model generating incoherent text. Lastly, we tried using pre-quantized models. We discovered that `onnx-community` provided these versions for the 3 smallest models of the `Qwen3` family. The first two worked, `onnx-community/Qwen3:4B` produced incoherent text. Each of these issues might have been solvable with a few hours of investment; however, solving all of them would have required a significant time commitment with no guarantee of success. Therefore, we decided to use full-precision models in our evaluation.

7.3.2 Experimental Setup Limitations

Network Our evaluation was carried out on a test bed where all participants are connected via LAN and in proximity. We observed latencies of less than 2 milliseconds and bandwidths of 10-100 MB/s. This led to computation dominating the inference time, even for configurations with 10 active workers. Evaluating our work in other settings, such as using WiFi, limiting bandwidth to 1 MB/s, or artificially introducing latency, would make our results more generalizable for practical deployments. However, we believe that such artificial network conditions can usually be approximated through calculations,

yielding limited benefits.

Hardware Our test bed includes 5 heterogeneous workers. Aidos uses 4 A4000 GPUs; the other devices are based on Jetson. The inability to use the GPU on the Jetson devices impacted more than half of our devices and our evaluation. We did not show the performance characteristics of our system using any GPU other than A4000. As a small upside, we were able to show that the system generalizes to any type of compute worker regardless of accelerator.

Model Download During experiments, all model weights were downloaded in advance to avoid affecting the partitioning strategy or negatively impacting bandwidth. Partitioning the model based on already downloaded weights is essential for real-world deployments. When the server assigns a model and enters the *Preparing* state, multiple workers might have to download many GB of data. Ensuring that all workers take the same time to download data is essential to serve requests as soon as possible. We were unable to verify how the server behaves in such scenarios, and are unable to draw conclusions for practical deployments with ephemeral workers that do not have all the weights pre-downloaded.

Workloads As our workloads, we use 3 different prompting strategies with a maximum sequence length of slightly more than 1024. We were able to show distinct performance characteristics across prompt types; however, large-scale models can easily exceed thousands of tokens in sequence length. We did not investigate the performance of prompts reaching such lengths.

Web-Based Inference We did not evaluate web-based inference in detail using more than one worker. This is due to two reasons: First, using Puppeteer did not work on Jetson devices. Secondly, since we were unable to test using web-based clients on Jetson, we decided against introducing further heterogeneity into our test bed. Since the server does not differentiate between worker types, only using their reported performance, there is no difference from the server's perspective between web-based and native workers. Although we cannot guarantee, we believe that web-based inference would have similar performance characteristics as native clients only with a slight decrease in computational speed.

7.4 Future Work

There are two main directions for future work that we wish to highlight. First, we will suggest immediate system improvements that represent more of an engineering problem than a research problem. Secondly, we will provide directions for further research that builds on this work.

7.4.1 System Extensions

Quantization Support Although we showed that we were able to run quantized models, providing a framework for easily converting and quantizing models is necessary to allow users to run new LLMs using our system. Evaluating the server using a broader range of quantized models would better position this work in the broader field of AI. Using quantized models allows for direct comparison against other inference engines.

Optimized Native Workers We used ONNX Runtime Web as our inference engine for web-based LLM inference. For native workers, this is not necessary. By using WebSockets for communication and ONNX as the model transport format, workers are free to run inference as they see fit. For simplicity, we used an almost one-to-one port of the TypeScript implementation for our native Python worker. Instead, we propose treating the ONNX model as just a transport format and implementing an optimized worker by converting the model. In theory, it should be possible to achieve performance close to engines such as vLLM using this approach with 1 worker. It is unclear how difficult it is to implement this in practice.

Adaptive Optimization For most trade-offs, the system is currently fixed in one direction. We propose a dynamic solution where the server makes decisions based on the environment and the connected workers. For example, in a low-bandwidth environment, the server enables zstd compression, or when working with higher churn rates, the server switches to KV cache syncing and waits for the inactive graph to commit before trying to find a better assignment.

Speculative Decoding Speculative Decoding is an inference optimization that uses a smaller model to generate multiple future tokens. The main model validates the prepared tokens in parallel. If the token predictions of the two models match, we can predict multiple tokens with a single forward pass. For scenarios where the main model is too large to run on one device, but smaller models can be deployed, speculative decoding might yield great performance improvements, especially since this saves extra round-trips, which would accumulate network overhead.

Chat Conversation In its current state, the server only supports simple 1-question prompts. Extending the server to support full chat conversions requires only a small time investment but adds significant practicality.

7.4.2 Research Questions

P2P vs. Star Topology

Using peer-to-peer communication would remove the server as the central communication bottleneck. The workers could communicate directly with each other, reporting only the next generated token back to the server. WebRTC enables peer-to-peer communication

in the browser. Using WebRTC, the server could still act as a central orchestrator. After partitioning the model, the server would "launch" a new group of workers to run distributed inference over WebRTC. The workers would only send back generated tokens and metrics, freeing the server from forwarding tensors and removing it as a networking bottleneck.

In theory, this approach is highly scalable. Since the server plays no role in communicating large amounts of data, it can manage thousands of concurrently connected workers serving many models in parallel.

Such a system could scale across the internet similarly to Petals [BRC⁺23], with the added benefit of supporting zero-cost deployment via the browser.

Multi Graph Partitioning

Tied to the idea of peer-to-peer communication using WebRTC is multi-graph partitioning. Currently, the server supports only one active graph, limiting its ability to scale across numerous workers. The difficulty of this problem lies in the graph partitioning algorithm. The algorithm would need to include information about all connected users and whether they are part of an active assignment. When combined with peer-to-peer approaches, it may be necessary to include additional information about network conditions between participants, which can lead to the Traveling Salesman Problem.

Non-LLM Models

In this work, we focus purely on LLM inference. Our choice of ONNX as the model format allows extension to other inference problems. For example, large-scale computer vision models could benefit from distributed inference. Extending the server to support more general distributed inference using ONNX offers a parallel use case to LLM inference.

Conclusion

In this thesis, we design an architecture for web-based distributed LLM inference. We use ONNX as the model format and ONNX Runtime Web as the inference engine in the browser. To facilitate heterogeneous workers, we use a dynamic partitioning algorithm to assign parts of the models to workers according to their capabilities. For communication, we use WebSockets for bidirectional communication and Protocol Buffers for binary encoding of messages and tensors.

We implement the inference server in Rust, providing an API for workers to connect and users to send inference requests. The server is responsible for orchestration; it manages connected workers, collects metrics, and automatically partitions the ONNX graph to minimize end-to-end latency during token generation. We implement two workers: a web-based worker running in the browser with TypeScript and WebGPU, and a native worker running with Python and CUDA.

We found that the server adds virtually no overhead during distributed token generation, taking up a maximum of 0.15% of time during inference. The network time remained below 25% for all requests. This suggests that the system can scale to large models with most of the time spent on computation rather than overhead.

Regarding network optimization, we found that different optimizations have asymmetric cost-benefit profiles. Moving token decoding into the ONNX graph provides the greatest benefit, reducing transferred bytes by 75MB per request with negligible overhead. Compression using zstd proved counterproductive on our fast network, introducing computational overhead that outweighed bandwidth savings. Prefix caching reduces TTFT by 25-30% for our 14-token system prompt, with benefits scaling for longer contexts.

The inference server is robust against arbitrary worker disconnects. Upon worker disconnect, it automatically repartitions the ONNX model, assigns the partitions to the workers, and resumes token generation. For configurations with 1 active worker, we found that the server retains more than 70% uptime even with a mean time to failure of

8. CONCLUSION

10 seconds. For models requiring 5 workers, the server struggles to remain operational. This is in line with our design goals of allowing for arbitrary disconnects but assuming relatively stable workers.

As our primary contribution, we demonstrate that distributed LLM inference on heterogeneous consumer hardware is viable for exploratory use cases. While production ready inference engines such as Ollama achieve 6-7x higher throughput than ONNX Runtime through quantization and optimized kernels, our system enables access to models beyond individual hardware capacity. Using the browser as a distribution tool, we achieve zero-setup deployment.

Overview of Generative AI Tools Used

LLMs were used during the development of the distributed inference server. We primarily used Gemini to better understand the complex ecosystem of ONNX and LLM inference.

During the writing of this thesis, LLMs were used sparingly. Gemini 3 Pro and Claude Sonnet 4.5 were used to refine the thesis's editing. Example prompts include:

```
<2 drafts generated by Claude based on earlier context>

I have these two draft of my discussion section in my master
thesis.
Both have their advantages and disadvantages and are not quite
right.
Merge the two so the result is a discussion that finishes off
my thesis nicely.
Start with Interpretation of Result and then go over to
research questions.
I believe a section about design tradeoffs fits well and
practical deployment also makes sense. Limitations should be
present in some way.
I also want a detailed future work section.
```

Resulting in the following (shortened to fit here):

```
\chapter{Discussion}\label{chap:discussion}

\section{Interpretation of Results}
Synthesize findings across all RQs with concrete insights
Connect quantitative results to practical implications

\subsection{RQ1: Zero-Setup Distribution via Web Technologies}
\begin{quote}
How can web technologies enable zero-setup distributed LLM
inference across heterogeneous consumer hardware?
```

```
\end{quote}

Achievement of zero-setup goal with measured trade-offs
Web workers enable zero-setup but sacrifice performance (33%
  slower for short sequences)
Web workers match native for long sequences - WebGPU maturity
  implications
Browser constraints encountered: memory limits, precision
  support, VRAM detection
Hybrid approach (web + native workers) provides flexibility
Zero-setup advantage outweighs performance penalty for
  exploratory use

...

\section{Future Work}
Research directions opened by this work
Organized by timeline: immediate extensions -> long-term vision

\subsection{Immediate System Extensions}
Incremental improvements addressing known limitations

...
```

The resulting outputs were not used 1 to 1, always requiring modifications. However, they served as great sources for inspiration, especially when rerunning prompts and merging their different outlooks on ideas.

To rewrite paragraphs, LLMs were used only in individual cases to continue the flow of writing without getting stuck on formulation.

List of Figures

2.1	A single layer of Qwen3-0.6B stored as an ONNX file and visualized using Netron.	11
4.1	The system’s star topology during inference, illustrating the centralized coordination of the server.	19
4.2	Distributed Autoregressive Token Generation	21
4.3	The minimum required communication for token generation from the server’s perspective.	27
5.1	Control-flow diagram of the dynamic repartitioning loop.	30
5.2	Event loop state transitions. The server uses a background staging variable (<i>inactive_graph</i>) to allow transitions from <i>Up</i> directly to <i>Committing</i> (Hot Swap), minimizing downtime during model updates.	37
6.1	Comparison of inference engines in terms of TPS for different prompting strategies.	45
6.2	Comparison of workers using the <i>chat</i> prompt with Qwen3:0.6B.	46
6.3	Comparison of models for across number of workers and metrics.	47
6.4	Absolute and relative time taken by each component in the system.	49
6.5	Comparison of optimization strategies in terms of TTFT and TPOT. For the baseline all optimizations are enabled. Otherwise a single optimization is disabled at a time.	52
6.6	Exponentially Weighted Moving Average (EWMA) of TPS overlaid with server state changes.	55
6.7	The proportion of time the server spends in each state aggregated using a rolling window of 120 seconds.	56

List of Tables

2.1	The model size in bytes for different models. The model size for full precision was taken from Huggingface [Hug25a], for the quantized model size we used Q4_K_M GGUF from the Ollama model library [Oll].	8
3.1	Comparison of LLM inference systems	16
4.1	Comparison of bandwidth costs between KV cache recomputation and synchronization. Initial Size refers to the KV cache size of the prompt. Slope(Rec) denotes the network transfer cost of activations between workers per generated token. Slope(Sync) denotes the cost per token to synchronize the new KV cache slice and the recovery cost for transferring the cache to a replacement worker. Equilibrium is the sequence length x where the recomputation cost ($k_{rec}x + d_{rec}$) equals the total synchronization cost ($k_{sync}x + d_{prompt}$). Values of <i>Recompute</i> or <i>Sync</i> indicate strict superiority of a strategy. <i>Note:</i> In practice, if generation finishes before equilibrium is reached, recomputation is superior as it avoids the continuous overhead of synchronization. Small tensors and model I/O are excluded; logits are optimized via elimination (see Section 5.2.2).	23
6.1	Device Hardware Specifications	39
6.2	Worker assignments for each experiment configuration	39
6.3	Worker Network Statistics	40
6.4	Characteristics of ONNX models. Models were converted using Microsoft Olive.	41
6.5	Network Stats (bytes) for Qwen3-0.6B	50
6.6	Network Stats (bytes) for Qwen3-1.7B	51
6.7	Network Stats (bytes) for Qwen3-4B	51
6.8	Fault Tolerance Statistics	57
6.9	Fault Tolerance Statistics	57
6.10	State Transitions - Qwen3-0.6B, chat	57
6.11	State Transitions - Qwen3-1.7B, chat	58
6.12	State Transitions - Qwen3-4B, chat	58

List of Algorithms

5.1	Dynamic Programming for Layer-to-Worker Assignment	32
-----	--	----

Bibliography

- [AMY18] Fan Angela, Lewis Mike, and Dauphin Yann. Hierarchical Neural Story Generation. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018.
- [aph25] Aphrodite-engine/aphrodite-engine. <https://github.com/aphrodite-engine/aphrodite-engine>, December 2025.
- [BMR⁺20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.
- [BOV24] Felix Brakel, Uraz Odyurt, and Ana-Lucia Varbanescu. Model Parallelism on Distributed Infrastructure: A Literature Review from Theory to LLM Case-Studies, March 2024.
- [BRC⁺23] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and Colin A. Raffel. Distributed Inference and Fine-tuning of Large Language Models Over The Internet. *Advances in Neural Information Processing Systems*, 36:12312–12331, December 2023.
- [Chr] Chromium Docs - Sandbox. <https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md>.
- [CMJ⁺18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 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.

- [CMSL25] Zhiyang Chen, Yun Ma, Haiyang Shen, and Mugeng Liu. WeInfer: Unleashing the Power of WebGPU on LLM Inference in Web Browsers. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, pages 4264–4273, New York, NY, USA, April 2025. Association for Computing Machinery.
- [Dec25] Decoding Strategies in Large Language Models. <https://huggingface.co/blog/mlabonne/decoding-strategies>, September 2025.
- [dee25] Deepspeedai/DeepSpeed. <https://github.com/deepspeedai/DeepSpeed>, December 2025.
- [DFE⁺22] 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, December 2022.
- [DLBZ22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, December 2022.
- [DLF⁺24] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huaqian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yudian Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen

Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model, June 2024.

- [DLM⁺25] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bower Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y. Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yudian Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin

Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models, December 2025.

- [Epo25] Epoch AI. Data on AI Models, July 2025.
- [exo25] Exo-explore/exo. <https://github.com/exo-explore/exo>, December 2025.
- [fac25] Facebook/zstd. <https://github.com/facebook/zstd>, December 2025.
- [FAHA23] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers, March 2023.
- [FZS22] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.
- [ggm25] Ggml-org/llama.cpp. <https://github.com/ggml-org/llama.cpp>, December 2025.
- [GHA23] Hock-Ann Goh, Chin-Kuan Ho, and Fazly Salleh Abas. Front-end deep learning web apps development and deployment: A review. *Applied Intelligence*, 53(12):15923–15945, June 2023.
- [HBD⁺20] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The Curious Case of Neural Text Degeneration. In *Eighth International Conference on Learning Representations*, April 2020.
- [HCB⁺19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [HIZ⁺22] Yang Hu, Connor Imes, Xuanang Zhao, Souvik Kundu, Peter A. Beerel, Stephen P. Crago, and John Paul Walters. PipeEdge: Pipeline Parallelism for Large-Scale Model Inference on Heterogeneous Edge Devices. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 298–307, August 2022.

- [Hug25a] Hugging Face – The AI community building the future. <https://huggingface.co/>, December 2025.
- [hug25b] Huggingface/optimum. <https://github.com/huggingface/optimum>, December 2025.
- [Imp] Implementation Status. <https://github.com/gpuweb/gpuweb/wiki/Implementation-Status>.
- [KLZ⁺23] 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 '23*, pages 611–626, New York, NY, USA, October 2023. Association for Computing Machinery.
- [LLM] LLM Inference guide for Web | Google AI Edge | Google AI for Developers. https://ai.google.dev/edge/mediapipe/solutions/genai/llm_inference/web_js.
- [LTT⁺24] 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 On-Device LLM Compression and Acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, May 2024.
- [LZP⁺24] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache, July 2024.
- [MF11] Alexey Melnikov and Ian Fette. The WebSocket Protocol. Request for Comments RFC 6455, Internet Engineering Task Force, December 2011.
- [mic25] Microsoft/Olive. <https://github.com/microsoft/Olive>, December 2025.
- [mic26] Microsoft/onnxruntime-genai. <https://github.com/microsoft/onnxruntime-genai>, February 2026.
- [MP23] Anthony Moi and Nicolas Patry. HuggingFace’s Tokenizers. <https://github.com/huggingface/tokenizers>, April 2023.
- [MXZ⁺19] Yun Ma, Dongwei Xiang, Shuyu Zheng, Deyu Tian, and Xuanzhe Liu. Moving Deep Learning into Web Browser: How Far Can We Go? In *The World Wide Web Conference, WWW '19*, pages 1234–1244, New York, NY, USA, May 2019. Association for Computing Machinery.

- [MY17] Freitag Markus and Al-Onaizan Yaser. Beam Search Strategies for Neural Machine Translation. *Proceedings of the First Workshop on Neural Machine Translation*, 2017.
- [Ngu25] Xuan-Son Nguyen. Ngxson/wllama. <https://github.com/ngxson/wllama>, December 2025.
- [NHP⁺19] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, pages 1–15, New York, NY, USA, October 2019. Association for Computing Machinery.
- [NVI25] NVIDIA/TensorRT-LLM. <https://github.com/NVIDIA/TensorRT-LLM>, December 2025.
- [Oll] Ollama Model Library. <https://ollama.com/library>.
- [oll25] Ollama/ollama. <https://github.com/ollama/ollama>, December 2025.
- [ONN18] ONNX Runtime developers. ONNX Runtime. <https://github.com/microsoft/onnxruntime>, November 2018.
- [onn25] Onnx/onnx. <https://github.com/onnx/onnx>, December 2025.
- [pro25] Protocolbuffers/protobuf. <https://github.com/protocolbuffers/protobuf>, December 2025.
- [RQZ⁺24] Charlie F. Ruan, Yucheng Qin, Xun Zhou, Ruihang Lai, Hongyi Jin, Yixin Dong, Bohan Hou, Meng-Shiun Yu, Yiyang Zhai, Sudeep Agarwal, Hangrui Cao, Siyuan Feng, and Tianqi Chen. WebLLM: A High-Performance In-Browser LLM Inference Engine, December 2024.
- [RWC⁺19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.
- [Sec] Security/Sandbox - MozillaWiki. <https://wiki.mozilla.org/Security/Sandbox>.
- [sgl25] Sgl-project/sglang. <https://github.com/sgl-project/sglang>, December 2025.
- [SPP⁺20] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, March 2020.

- [SYC⁺24] Chufan Shi, Haoran Yang, Deng Cai, Zhisong Zhang, Yifan Wang, Yujiu Yang, and Wai Lam. A Thorough Examination of Decoding Methods in the Era of LLMs, February 2024.
- [The25] The WebSocket API (WebSockets) - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API, December 2025.
- [tok25] Tokio-rs/tokio. <https://github.com/tokio-rs/tokio>, December 2025.
- [Tra] Transformers.js. <https://huggingface.co/docs/transformers.js/en/index>.
- [TTHT21] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. Automatic Graph Partitioning for Very Large-scale Deep Learning. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1004–1013, May 2021.
- [vll25] Vllm-project/vllm. <https://github.com/vllm-project/vllm>, December 2025.
- [VSP⁺17] 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*, volume 30. Curran Associates, Inc., 2017.
- [Weba] Web Neural Network API. <https://www.w3.org/TR/webnn/>.
- [Webb] WebGPU. <https://gpuweb.github.io/gpuweb/>.
- [Webc] WebGPU - Malicious. <https://gpuweb.github.io/gpuweb/#malicious-use>.
- [Webd] WebGPU - W3. <https://www.w3.org/TR/webgpu/>.
- [Web25] WebGPU API - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API, November 2025.
- [WJC⁺25] Qipeng Wang, Shiqi Jiang, Zhenpeng Chen, Xu Cao, Yuanchun Li, Aoyu Li, Yun Ma, Ting Cao, and Xuanzhe Liu. Anatomizing Deep Learning Inference in Web Browsers. *ACM Trans. Softw. Eng. Methodol.*, 34(2):47:1–47:43, January 2025.
- [YGW⁺25] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. EdgeMoE: Empowering Sparse Large Language Models on Mobile Devices. *IEEE Transactions on Mobile Computing*, 24(8):7059–7073, August 2025.

- [YJK⁺22] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.
- [YLY⁺25] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report, May 2025.
- [YWN⁺25] Zhengyi Yuan, Xiong Wang, Yuntao Nie, Yufei Tao, Yuqing Li, Zhiyuan Shao, Xiaofei Liao, Bo Li, and Hai Jin. DynPipe: Toward Dynamic End-to-End Pipeline Parallelism for Interference-Aware DNN Training. *IEEE Transactions on Parallel and Distributed Systems*, 36(11):2366–2382, November 2025.
- [ZLC⁺24] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuancheng Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.
- [ZLZ⁺22] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.
- [ZSC⁺25] Mingjin Zhang, Xiaoming Shen, Jiannong Cao, Zeyang Cui, and Shan Jiang. EdgeShard: Efficient LLM Inference via Collaborative Edge Computing. *IEEE Internet of Things Journal*, 12(10):13119–13131, May 2025.
- [ZSL⁺24] Junchen Zhao, Yurun Song, Simeng Liu, Ian G. Harris, and Sangeetha Abdu Jyothi. LinguaLinked: Distributed Large Language Model Inference on Mobile Devices. In Yixin Cao, Yang Feng, and Deyi Xiong, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 160–171, Bangkok, Thailand, August 2024. Association for Computational Linguistics.