PIPELINE PARALLELISM WITH SPECULATIVE DECODING BASED ON VLLM

Zepeng Zhao [*1] **Zhen Tong** [*1]

ABSTRACT

This work explores the integration of pipeline parallelism and speculative decoding in vLLM V1, a modern inference and serving framework for large language models (LLMs). We implemented intra-node pipeline parallelism using multiprocessing and evaluated two architectural designs for combining speculative decoding with pipeline execution. Our findings indicate that careful placement of the drafter model and communication strategies significantly impact system performance. Our evaluation of 3D parallelism configurations also reveals trade-offs between tensor-parallelism and pipeline-parallelism depending on model size and speculative decoding settings.

1 INTRODUCTION

vLLM is a fast and user-friendly library designed for LLM inference and serving. It has evolved from a proof-of-concept into a community-driven project with active contributions from both academia and industry. Speculative decodingOne is a optimization techniques for improving inference speed, initially proposed by Google. This method employs a lightweight drafter model to predict multiple tokens ahead, which are then verified by a larger verifier model, reducing overall latency. With the release of vLLM V1, the project introduces a cleaner and more performant codebase compared to V0. However, some core features such as intra-node pipeline parallelism for speculative decoding remain underdeveloped, requiring new system-level integration and engineering effort.

2 BACKGROUND

The goal of pipeline parallelism is to divide a large model across multiple processes or devices, allowing concurrent execution of different layers to increase throughput. In speculative decoding, the drafter model generates a batch of tokens which are later verified by the full model. While effective, this mechanism introduces complexity when integrated with pipeline execution due to the interaction between model layers and shared weights, especially in architectures like Eagle3, where the embedding layer is borrowed from the verifier model.

EAGLE-3 EAGLE-3(Li et al., 2025) is a latest enhanced speculative-decoding framework built on top of EAGLE-2(Li et al., 2024). EAGLE-3 makes two key architectural changes: it removes the feature-prediction constraint and instead performs a training-time test that directly predicts draft tokens, and it fuses low-, mid-, and high-level features (l, m, h) from the target model (rather than relying solely on top-layer embeddings) to provide richer context to the drafter :contentReference[oaicite:0]index=0. The three vectors l, m, h are concatenated and projected into a unified feature g before being fed into the drafter decoder. Moreover, by adopting the dynamic draft-tree pruning mechanism from EAGLE-2, EAGLE-3 adjusts its speculative tree based on model confidence to further boost acceptance rates and speedup. Empirically, EAGLE-3 achieves average acceptance lengths of around 5–6 tokens (e.g. $\tau \approx 5.88$ for LLaMA-3.3 70B) and speedup ratios up to 6.5× across benchmarks, while also improving throughput in large-batch regimes (up to 1.75× over vanilla vLLM at batch 56).

vLLM In the work of vllm(Kwon et al., 2023), PagedAttention was introduced as an attention mechanism inspired by operating-system paging that partitions each request's KV cache into fixed-size "pages" and allows these pages to be shared across requests and sequences. This design eliminates memory fragmentation and enables flexible KV cache management. Building on this, they implemented a centralized scheduler and a block-level memory manager in the vLLM engine, achieving nearly zero waste of GPU KV cache and a 2–4× throughput improvement over prior systems such as Orca(Yu et al., 2022) across a variety of models and tasks, while maintaining comparable latency.

vLLM V1 vs. V0 vLLM V1 is a ground-up re-architecture of the original vLLM V0 engine. While V0 proved the benefits of paged KV-cache management, its feature set grew organically—leading to technical debt and a fragmented

[*]Equal contribution [1]Carnegie Mellon University. Correspondence to: Tianqi Chen <15642 Course Staff>.

codebase. V1 retains V0's battle-tested components (models, GPU kernels, utilities) but rewrites the core scheduler, KV cache manager, worker, sampler, and API server into a single, modular engine. All optimizations (prefix caching, chunked prefill, speculative decode, etc.) are enabled by default, yielding near-zero CPU overhead and much simpler code.

Speculative Decoding in V0 vs. V1 vLLM V0 shipped with several speculative decoding schemes—such as Medusa (Cai et al., 2024) and MLP-style speculative decoding (Wertheimer et al., 2024)—none of which are available in V1. Furthermore, neither V0 nor V1 supports pipeline parallelism together with any speculative decoding mode. In V1, only the ngram-based and Eagle modes are temporarily supported: the ngram mode uses a KMP string-matching algorithm for draft proposals rather than the model itself, and the Eagle mode implements only the Eagle-2 design (Eagle-3 is not supported).

3 System Design

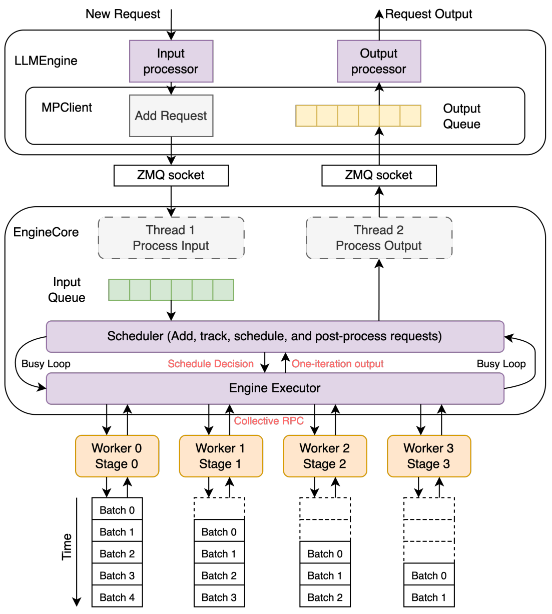


Figure 1. vLLM framwork overview

3.1 Project Scope

With limited computing resource access, our work focuses on a single node with multiple GPUs:

- Implementing intra-node pipeline parallelism within vLLM V1 engine using multiprocessing framework and built-in data transfer tools. We don't support Ray because it targets distributed inference scenario.
- Extending speculative decoding features in vLLM V1 engine to make it compatible with any arbitrary pipeline parallel setups.
- Empirically studying the impact of 3D parallelism strategies (data parallelism, pipeline parallelism, and tensor parallelism) on the performance of the inference system.

3.2 vLLM V1

vLLM V1, shown in Figure 1, provides a simple, modular, and easy-to-hack codebase, ensuring high performance with near-zero CPU overhead. It separates the front end and backed end to provide flexibility and enable online serving. vLLM V1 introduces a comprehensive re-architecture of its core components, including the scheduler, KV cache manager, worker, sampler, and API server. However, it still shares a lot of code with vLLM V0, such as model implementations, GPU kernels, distributed control plane, and various utility functions.

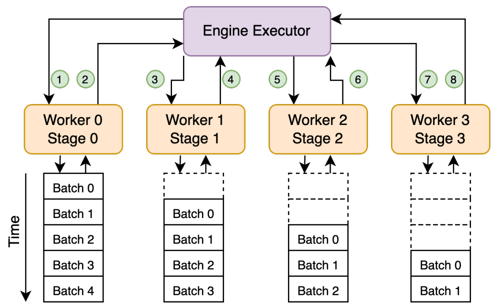
The front end LLMEngine handles all incoming raw requests and packages completion outputs. For single modality LLMs, i.e., our scenario, it runs the tokenizer to get token ids, and maps the output token ids back to human-readable texts. It runs a busy loop inside to poll the new requests from the entry point and finished requests from the Engine Core.

At the back end, Engine Core manages all the computation details, like launching the workers, scheduling the requests, managing KV caches, and dispatching batches. It also runs a busy loop inside to repeatedly process new requests, call the scheduler to schedule new requests, send decisions to the executor to dispatch batches to workers, and process the per-iteration results received from the workers.

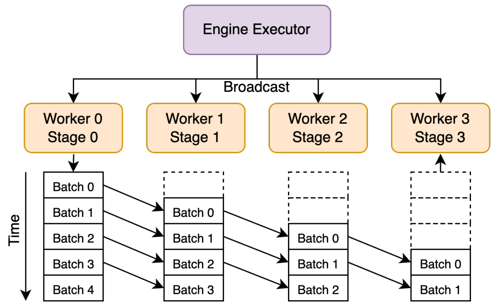
To realize pipeline parallelism, we created a new version of Engine Executor and Scheduler which can be enabled by user configuration. Our scheduler takes the on-the-fly requests into consideration so that they won't be scheduled unexpectedly. And our engine executor adapts the same API as other executors in the framework, but uses asynchronous collective RPCs under the hood to support pipeline parallelism. These modules are very flexible and completely compatible with the downstream and upstream parts. When user specifies pipeline parallel size greater than one, the framework will instantiate our modules instead of the default ones, thus achieving seamless adaptation.

3.3 Pipeline Parallelism Architecture

We designed and tested two different pipeline execution strategies. In the centralized design, as shown in Figure 2a, the Engine Executor manages both scheduling and inter-stage communications. While relatively simple and easy



(a) Centralized Design



(b) Decentralized Design

Figure 2. Two pipeline designs

to implement, this approach incurs high overhead due to repeated communication between the engine and all worker processes. This design serves as the baseline for us to verify the correctness of model partitioning, weights loading and inter-stage communications.

On the other hand, in the decentralized design, as shown in Figure 2b, upon every request batch scheduled by the system, the Engine Executor broadcasts this information to all stages asynchronously and waits only for the output from the stage. All workers communicate directly with each other, alleviating the burden of the Engine Executor and reducing unnecessary transfer of hidden states. This design significantly reduced inference latency.

To fully exploit the potential of pipeline parallelism, we should also dispatch multiple batches into the pipeline to saturate it, which requires the cooperation of the system scheduler. The scheduler should recognize the requests that are on the fly and avoid scheduling them again before their output of last scheduling is retrieved. And the Engine Executor should also utilize asynchronous message broadcast and remember the batches dispatched. We implemented an asynchronous version of both to achieve this.

3.4 Speculative Decoding with Pipeline Parallelism

We explored two placement strategies for integrating the drafter model into a pipeline-fashioned execution workflow, like Figure 3. Intuitively, placing the drafter before the pipeline allows its outputs to naturally flow into the full model for verification. However, vLLM currently employs a special design forcing the rejection sampler and the drafter model to be placed together and hard to decouple. Considering this constraint, this placement forces the last stage to send the whole final logits back to the first stage for it to speculate new tokens, increasing communication cost. By placing the drafter with the final pipeline stage, it can receive logits directly and avoid feedback loops. This design reduces overhead and simplifies KV cache synchronization.

There is one drawback of placing the drafter model at the very last stage. Since the drafter accepts token id, instead of logits, from the output of the rejection sampler, it needs the embedding layer to process token id into hidden states. If placed on the last stage, we should load another copy of the embedding layer dedicated for the drafter model. But with the very limited size of embedding layer, this extra overhead can be ignored.

The *Eagle3* speculative decoding algorithm requires the hidden states from the middle layers for the model. For example, in *Llama3.1-8B-Instruct*, it requires the hidden states output from layer 2, 19 and 29. When the pipeline parallel size is 1, these hidden states are local to the drafter model so can be easily preserved. When we partition the model into separate stages, we should handle these middle-layer hidden states specially, so that they can be correctly passed into the last stage and used by the drafter model.

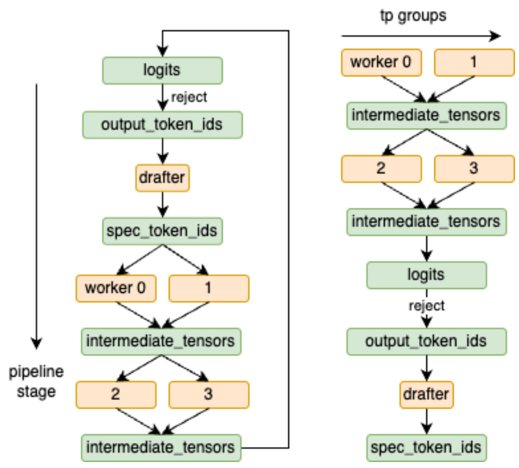


Figure 3. Drafter design

Table 1. Sampling parameters

Sampling Parameter	Value
Max model length	2048
Max num sequence	1024
Enforce eager mode	True
Temperature	0
Global seed	10086
Max token per request	256
Distributed backend	multiprocessing
Torch distributed backend	NCCL

4 EVALUATION

4.1 Evaluation Setup

We did the experiments with latest *Eagle3* model, *LLaMA-Instruct 3.1 8B* and *LLaMA-Instruct 3.3 70B*(Li et al., 2025).

For datasets, we choose ones used by Llama Eagle3 to provide better comparison, including the MT-bench(Zheng et al., 2023), HumanEval (Chen et al., 2021), GSM8K (Cobbe et al., 2021), Alpaca (Zhang et al., 2025), CNN/Daily Mail (See et al., 2017)

Our benchmarks are performed on a single machine with 2 AMD EPYC 9534 CPUs and 8 H100 Nvidia GPUs. Due to very limited access to these expensive computing resources, data are collected on a one-shot basis instead of the average of multiple rounds of tests. We use vLLM 0.8.5 as our startup codebase, and Torch 2.6.0, Triton 3.1.0, CUDA 12.2, NCCL 2.13.0, and Python 3.12 as our software environment.

Table 1 shows the parameters we set for our experiments. These parameters are set w.r.t. to our hardware capacity and the software versions we use.

4.2 Speculative Decoding

Analysis throughput improvement, and acceptance rate.

Table 3 shows our testing results under the combination of speculative decoding and pipeline parallelism, based on *Llama3.1-8B-instruct* model. For stages varying from 1 to 4, the average acceptance rate (AR) and average acceptance length (AL) remains almost the same, showing good compatibility between these two features. And the acceptance rate and acceptance length of EAGLE3 is significantly better than EAGLE1, proving the correctness of our implementation.

However, as the data shows that with EAGLE1 and EAGLE3, the throughput isn't comparable with the one of disabling speculative decoding. According to the original paper(Li et al., 2025), this is the case as expected. The papers states that at batch size of 56, the throughput of performance gain is near 0, and our setup uses batch size of 1024. Increasing the batch size dramatically hurts the performance of EAGLE3, resulting in the worse overall throughput. However, as the latency column shows, with only 4 sequences or batch size, the latency of EAGLE3 is always the lowest, as expected from the original paper.

The reason why the throughput doesn't scale with the number of pipeline stages is that when collecting these data, we had not finished the asynchronous version of the pipeline parallelism, i.e., forcing synchronization between stages for each batch. The stability of throughput across different pipeline parallel size substantiates this point.

4.3 3D Parallelism Exploration

Table 3 shows our experiments on different 3D parallelism strategies, with *Llama3.1-70B-instruct* and *Llama3.1-8B-instruct* models. All unit tests are enabled with speculative decoding, specifically, *Llama3.1-70B-instruct* is tested with EAGLE1 and *Llama3.1-8B-instruct* is tested with EAGLE3.

For big models that cannot fit into a single GPU due to excessive model size, like *Llama3.1-70B-instruct*, at bare minimum, we should launch it with 4 GPUs. Since data parallelism doesn't help with big models, we must rely on pipeline parallelism and tensor parallelism to shard the model weights. The highest throughput and lowest latency result from the combination of (1, 1, 8). This can be explained by noting the sheer size of this model (requiring at least 4 GPUs to load) and the efficiency of tensor parallelism in a single node. With the help of NVLink, collective operations are super fast, significantly facilitating tensor parallelism.

For small models that can be easily fit into a single GPU, like *Llama3.1-8B-instruct*, results are slightly different. As our expectation, the highest throughput should be produced by (8, 1, 1), since this combination minimizes the communication between GPUs and therefore has the lowest latency. However, as our test result shows, (1, 1, 8) gives us the greatest throughput while (8, 1, 1) is almost the worst of all cases. Based on our understanding of the system, we believe that this can be explained by the CPU contention between different data-parallel instances. When launching data-parallel instances, different ranks doesn't know the existence of each other, resulting in a contention for all CPU threads by the usage of Torch. If we could set a proper value for all data-parallel instances, the performance of data-parallalized combinations should show better results.

The biggest insight we have here is that, for inference in a single node, adding tensor parallel size is never a bad idea, even though diminishing returns are always observed across all cases. And pipeline parallelism and data parallelism are not the best choice for a single node, they are much better

Table 2. Performance of different pipeline parallelism across benchmarks for Eagle-Style Speculative Decoding

Model-SD	Configuration			mt_bench				openai_humaneval				gsm8k				alpaca				cnn_dailymail			
	dp	pp	tp	AR	AL	TP	Lat	AR	AL	TP	Lat	AR	AL	TP	Lat	AR	AL	TP	Lat	AR	AL	TP	Lat
`llama8B-None`	1	1	1	NA	NA	17662.95	2.3931	NA	NA	17095.76	2.3768	NA	NA	14708.28	1.5437	NA	NA	12590.01	2.2664	NA	NA	7897.03	2.3959
	1	2	1	NA	NA	17175.38	2.6721	NA	NA	17303.33	2.6438	NA	NA	14611.95	1.4674	NA	NA	12370.55	2.5517	NA	NA	9173.21	2.6692
	1	4	1	NA	NA	16434.53	3.7006	NA	NA	17262.86	4.0829	NA	NA	14303.05	2.2865	NA	NA	12293.34	4.0251	NA	NA	9158.36	4.0833
	1	2	2	NA	NA	22037.01	3.1181	NA	NA	22999.06	3.2421	NA	NA	19030.17	1.8822	NA	NA	16789.80	3.0432	NA	NA	13377.18	3.1065
`llama8B-eagle`	1	1	1	0.3524	2.2742	8128.61	2.1927	0.4524	3.1465	10167.45	1.3069	0.3962	2.7067	8182.57	1.0090	0.3708	2.4886	6994.48	1.6371	0.3334	2.2829	5258.47	2.0646
	1	2	1	0.3525	2.2906	8101.23	2.3980	0.4524	3.1490	10332.06	1.4591	0.3962	2.6916	8215.91	1.0965	0.3715	2.5687	7070.02	1.6631	0.3337	2.2609	5668.19	2.1975
	1	4	1	0.3525	2.2816	8135.37	2.6960	0.4524	3.1537	10570.92	1.6195	0.3962	2.6942	8496.61	1.2458	0.3707	2.5439	7166.88	1.8404	0.3339	2.3403	5783.80	2.1594
	1	2	2	0.3517	2.2978	10779.99	2.5795	0.4527	3.1922	14210.09	1.7254	0.3963	2.7079	11587.03	1.1749	0.3694	2.5419	9542.09	2.0411	0.3342	2.2548	8093.95	2.5701
`llama8B-eagle3`	1	1	1	0.5154	3.3447	11724.66	1.3701	0.6498	4.4069	14337.97	0.9761	0.5659	3.8558	11266.97	0.6070	0.5679	3.8490	9687.47	1.0118	0.5143	3.4716	7182.06	1.3757
	1	2	1	0.5154	3.3569	11641.57	1.4452	0.6498	4.4081	14374.24	1.0679	0.5658	3.8733	11306.92	0.6674	0.5681	3.9387	9858.20	0.8929	0.5134	3.3930	7638.41	1.4985
	1	4	1	0.5153	3.3598	11556.87	1.6513	0.6498	4.4070	14802.28	1.1989	0.5660	3.9151	11608.89	0.7480	0.5683	3.9676	9964.95	1.0079	0.5135	3.3900	7778.58	1.6195
	1	2	2	0.5135	3.3327	15518.51	1.9113	0.6507	4.3715	20104.27	1.3782	0.5648	3.8987	15497.52	0.8172	0.5668	3.7192	15497.52	1.4763	0.5142	3.3579	11063.44	1.9128

Table 3. Performance of different 3D-parallel configurations across benchmarks

Model-SD	Configuration			mt_bench				openai_humaneval				gsm8k				alpaca				cnn_dailymail			
	dp	pp	tp	AR	AL	TP	Lat	AR	AL	TP	Lat	AR	AL	TP	Lat	AR	AL	TP	Lat	AR	AL	TP	Lat
`llama3-70B-eagle`	1	1	8	0.2643	1.7547	4131.19	5.9241	0.3283	2.3195	5124.71	4.0056	0.2831	2.0345	4008.95	2.3784	0.2780	1.9314	3595.84	4.9302	0.2610	1.7800	3003.48	5.2179
	1	2	4	0.2644	1.7441	2673.50	6.2046	0.3283	2.3109	3340.88	4.5031	0.2834	2.0323	2626.04	2.5720	0.2782	1.9305	2391.47	4.8634	0.2611	1.7886	1968.18	5.5995
	1	4	2	0.2644	1.7614	1620.85	6.9488	0.3287	2.3007	2021.28	5.1158	0.2839	2.0321	1593.62	3.0472	0.2779	1.9198	1442.03	5.9690	0.2606	1.7828	1166.70	6.5717
	1	8	1	0.2648	1.7522	941.11	11.4179	0.3275	2.2790	1158.59	8.7999	0.2835	2.0299	915.87	5.0895	0.2773	1.9355	830.92	9.8312	0.2609	1.7882	668.91	10.9296
	2	1	4	0.2606	1.7083	2557.69	5.5794	0.3093	1.9858	2873.84	4.2392	0.2753	1.9125	2349.58	2.4337	0.2772	1.9357	2264.78	4.7250	0.2600	1.8190	1708.77	5.2671
	2	2	2	0.2609	1.6830	1546.70	7.4731	0.3122	1.9883	1793.06	5.3387	0.2766	1.9173	1449.22	3.0259	0.2777	1.9335	1382.54	5.8844	0.2600	1.8123	1064.78	6.5459
	2	4	1	0.2560	1.6393	869.62	12.1140	0.3045	1.9719	1004.93	8.5786	0.2712	1.9280	799.55	5.0285	0.2781	1.9175	803.78	9.7958	0.2602	1.8090	609.01	11.2168
`llama3-8B-eagle3`	1	1	1	0.5154	3.3447	11724.66	1.3701	0.6498	4.4069	14337.97	0.9761	0.5659	3.8558	11266.97	0.6070	0.5679	3.8490	9687.47	1.0118	0.5143	3.4716	7182.06	1.3757
	1	1	2	0.5135	3.3476	16094.30	1.7443	0.6507	4.3784	20205.60	1.2889	0.5648	3.8897	15852.93	0.7493	0.5665	3.7797	13584.69	1.3405	0.5137	3.3602	11024.28	1.8037
	1	1	4	0.5136	3.3275	20980.57	1.7610	0.6479	4.3491	26361.79	1.2422	0.5639	3.9265	21213.37	0.7361	0.5693	3.7069	18613.57	0.9726	0.5138	3.4303	15317.39	1.6582
	1	1	8	0.5141	3.3116	24647.14	1.7465	0.6485	4.4565	31102.90	1.2071	0.5658	3.9146	24550.42	0.7271	0.5671	3.5561	21280.96	1.0783	0.5146	3.3641	19343.24	1.6172
	1	2	1	0.5154	3.3569	11641.57	1.4452	0.6498	4.4081	14374.24	1.0679	0.5658	3.8733	11306.92	0.6674	0.5681	3.9387	9858.20	0.8929	0.5134	3.3930	7638.41	1.4985
	1	2	2	0.5135	3.3327	15518.51	1.9113	0.6507	4.3715	20104.27	1.3782	0.5648	3.8987	15497.52	0.8172	0.5668	3.7192	13809.18	1.4763	0.5142	3.3579	11063.44	1.9128
	1	2	4	0.5136	3.3082	18539.47	1.7965	0.6483	4.4143	25933.18	1.3127	0.5639	3.8536	19797.26	0.8450	0.5665	3.7325	17796.60	1.4252	0.5148	3.3145	15323.58	1.7016
	1	4	1	0.5153	3.3598	11556.87	1.6513	0.6498	4.4070	14802.28	1.1989	0.5660	3.9151	11608.89	0.7480	0.5683	3.9676	9964.95	1.0079	0.5135	3.3900	7778.58	1.6195
	1	4	2	0.5138	3.3577	13813.92	2.1881	0.6506	4.3669	19498.03	1.5484	0.5650	3.9040	15340.32	0.9026	0.5667	3.7096	13685.82	1.1903	0.5136	3.4458	10980.21	1.8400
	1	8	1	0.5154	3.3277	10763.10	2.8786	0.6499	4.4227	14550.69	1.9514	0.5654	3.9915	11467.36	1.1774	0.5682	3.8140	9910.21	1.9293	0.5133	3.3870	7845.51	2.5329
	2	1	1	0.5153	3.3167	11696.56	1.3315	0.6498	4.3568	14012.60	0.9712	0.5657	3.9115	11097.39	0.6082	0.5680	3.8885	9729.91	1.0387	0.5140	3.4008	7009.30	1.3678
	2	1	2	0.5133	3.2776	16001.63	1.7232	0.6508	4.3286	19477.50	1.2499	0.5647	3.7996	15612.48	0.7410	0.5661	3.7564	12302.48	1.3470	0.5140	3.3900	10807.04	1.6203
	2	1	4	0.5127	3.2880	21170.44	1.8752	0.6486	4.3765	25574.67	1.2724	0.5648	3.8383	19430.28	0.7386	0.5678	3.9092	16780.19	1.1150	0.5130	3.4582	15160.03	1.5105
	2	2	1	0.5151	3.3142	10987.57	1.4602	0.6498	4.3486	14340.26	1.0522	0.5657	3.8933	11143.27	0.6409	0.5692	3.8995	9614.35	1.0550	0.5133	3.3473	7677.19	1.4852
	2	2	2	0.5136	3.3131	14890.53	1.9095	0.6509	4.3296	20050.50	1.3769	0.5643	3.7930	14801.55	0.8030	0.5663	3.8672	13053.40	1.2094	0.5140	3.3030	10906.09	1.8998
	2	4	1	0.5148	3.2857	10590.81	1.7092	0.6497	4.3859	14621.41	1.2022	0.5655	3.9245	11070.56	0.7471	0.5684	3.8739	9588.92	1.2362	0.5140	3.3290	7746.27	1.6121
	4	1	1	0.5154	3.2855	11137.28	1.3380	0.6496	4.3475	13217.98	0.9616	0.5655	3.8932	10327.09	0.6019	0.5686	3.9446	9264.57	0.8409	0.5131	3.3473	7292.48	1.3662
	4	1	2	0.5124	3.2895	14510.12	1.8731	0.6511	4.3261	18871.89	1.2789	0.5642	3.8008	13914.09	0.7428	0.5666	3.8279	11585.80	1.3372	0.5136	3.3052	10348.80	1.7443
	4	2	1	0.5152	3.3146	9951.43	1.4592	0.6497	4.3596	13754.61	1.0277	0.5656	3.8905	10568.36	0.6457	0.5683	3.8992	9106.07	1.1134	0.5131	3.3538	7498.40	1.4816
	8	1	1	0.5098	3.2324	10250.77	1.3580	0.6431	4.3170	12210.67	0.9896	0.5755	3.8616	10222.01	0.6018	0.5613	3.9447	7549.21	0.8425	0.5166	3.3455	7062.10	1.3960

Table 3. **Note:** *Model–SD* refers to the verification model and its speculative decoding method (SD). For each benchmark (MT-Bench, HumanEval, GSM8K, Alpaca, CNN/DailyMail), the metrics reported are: AR (Acceptance Rate), AL (Average Accepted Length), TP (Throughput in tokens/second), and Lat (Latency in seconds).

for a distributed inference scenario.

5 CONCLUSION

In this project, we have extended vLLM V1 with a robust pipeline-parallelism-compatible Engine Executor and Scheduler, and seamlessly integrated them with the EAGLE-style speculative decoding mechanism. By partitioning the model's layers into pipeline stages and overlapping computation and communication, our implementation reduces end-to-end latency and improves hardware utilization. Crucially, compatibility with EAGLE speculative decoding ensures that the performance gains of pipeline parallelism are preserved even when speculative tokens are generated and rescored, enabling low-latency, high-throughput inference for Llama series models. This proves our deep understanding of the concepts learned from the lectures and our engineering skills working for a big project. We achieved our 100% goal in our proposal.

To understand the trade-offs inherent in modern distributed training and inference, we systematically evaluated a suite of 3D parallelism, including data parallelism, pipeline parallelism, and tensor parallelism. Our experiments across multiple model sizes and setups revealed that:

1. With a single node, data parallelism may incur contention for CPU resources and pipeline parallelism isn't as good as tensor parallelism, if we have NVLink installed.
2. For large models, tensor parallelism and pipeline parallelism are the only choices we have to enable inference, and tensor parallelism gives the best performance.
3. For small models, tensor parallelism is also the best for intra-node parallelism. If we could have resolve the problem of CPU resource attention, we might have better result for data parallelism.

These results highlight that no single parallelism strategy universally outperforms all others; instead, the optimal configuration depends on model scale, GPU topology, and workload characteristics.

In summary, our contributions provide a new and robust implementation of pipeline parallelism with speculative decoding on the base of vLLM framework. We are considering making a pull request after we clean up the code and conduct further experiments. On the other hand, we believe the insights we gained through 3D parallelism exploration will help us deploy large models more efficiently and invoke deep thinking on the fundamentals and applications of these techniques.

References

Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., and Dao, T. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024. URL https://arxiv.org/abs/2401.10774.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle-2: Faster inference of language models with dynamic draft trees, 2024. URL https://arxiv.org/abs/2406.16858.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle-3: Scaling up inference acceleration of large language models via training-time test, 2025. URL https://arxiv.org/abs/2503.01840.

See, A., Liu, P. J., and Manning, C. D. Get to the point: Summarization with pointer-generator networks, 2017. URL https://arxiv.org/abs/1704.04368.

Wertheimer, D., Rosenkranz, J., Parnell, T., Suneja, S., Ranganathan, P., Ganti, R., and Srivatsa, M. Accelerating production llms with combined token/embedding speculators, 2024. URL https://arxiv.org/abs/2404.19124.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Zhang, X., Tian, C., Yang, X., Chen, L., Li, Z., and Petzold, L. R. Alpacare:instruction-tuned large language models for medical application, 2025. URL https://arxiv.org/abs/2310.14558.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.