arXiv:2501.14417v2 [cs.DC] 27 Jan 2025

DEEPFLOW: Serverless Large Language Model Serving at Scale

Junhao Hu[2*], Jiang Xu[1], Zhixia Liu[1], Yulong He[1], Yuetao Chen[1], Hao Xu[1], Jiang Liu[1], Baoquan Zhang[1], Shining Wan[1], Gengyuan Dan[1], Zhiyu Dong[1], Zhihao Ren[1], Jie Meng[1], Chao He[1], Changhong Liu[1], Tao Xie[2], Dayun Lin[1], Qin Zhang[1], Yue Yu[1], Hao Feng[1], Xusheng Chen[1†], Yizhou Shan[1†]

[1]*Huawei Cloud* [2]*Peking University*

Abstract

This paper introduces DEEPFLOW, a scalable and serverless AI platform designed to efficiently serve large language models (LLMs) at scale in cloud environments. DEEPFLOW addresses key challenges such as resource allocation, serving efficiency, and cold start latencies through four main design components. First, it uses a simple serverless abstraction called the request-job-task model, which helps manage AI workloads across post-training and model serving tasks. Second, it builds an in-house serving engine FLOWSERVE using a microkernel-inspired design, NPU-centric execution, and SPMD-based parallelism to optimize LLM serving. The system also includes novel scheduling policies tailored for both PD-disaggregated and PD-colocated configurations. With optimizations like pre-warmed pods, DRAM pre-loading, and NPU-fork, DEEPFLOW can scale up to 64 instances in seconds. DEEPFLOW has been in production for over a year, operating on a large Ascend NPU cluster and providing industry-standard APIs for fine-tuning, agent serving, and model serving to our customers.

1 Introduction

The rapid rise of generative AI, exemplified by the success of platforms like ChatGPT, has transformed the landscape of artificial intelligence and increased the demand for efficient, scalable systems capable of serving large language models (LLMs). Model-as-a-Service (MaaS) platforms, such as OpenAI's offerings, enable millions of users to access powerful AI capabilities through a simple API interface. Consequently, LLM serving has become one of the most crucial workloads in modern data centers.

Ensuring such services' performance, efficiency, and cost-effectiveness is a complex challenge, requiring advanced infrastructure and optimized systems. We identify the following challenges to achieve optimal performance and resource utilization while ensuring service level objectives (SLOs) for multi-tenant users. First, AI workloads vary significantly in duration, from fine-tuning that can last hours or even days, to agent serving and LLM serving, which typically range from seconds to minutes. This variation makes it difficult to dynamically allocate shared resources without either underutilizing or overloading the system. Second, as LLM serving becomes increasingly distributed and stateful, managing resource allocation, synchronization, and fault tolerance across multiple instances is more complex. A single inference request may span multiple distributed instances and involve cached states, creating additional challenges in ensuring efficient operation. Lastly, the variability in LLM serving demands leads to fluctuating resource needs, further complicating resource optimization and the handling of cold start latencies [8].

This paper presents DEEPFLOW, a system designed to address the challenges inherent in deploying and managing AI workloads at scale in cloud environments. DEEPFLOW is Huawei Cloud's fully-hosted and serverless platform, offering industry-standard fine-tuning, agent serving, and model serving APIs [25]. DEEPFLOW has been running for over a year atop a large Ascend NPU cluster [20].

We report how DEEPFLOW addresses the above challenges using four major design components: serverless abstraction and infrastructure, serving engine, scheduling algorithms, and scaling optimizations.

Serverless Infrastructure. At the core of DEEPFLOW is a serverless abstraction that simplifies AI workload management. Users send HTTP *requests* that trigger internal *jobs*, which are broken down into smaller *tasks*. This request-job-task abstraction allows for the dynamic scaling of workloads and efficient resource sharing across post-training and model-serving tasks. DEEPFLOW's architecture, as shown in Figure 1, is built around this serverless abstraction and consists of three core components: job executors (JEs), task executors (TEs), and a cluster manager. Job executors handle incoming requests and decompose them into manageable tasks that are then distributed to task executors for execution. The cluster

*This work was completed during his internship at Huawei Cloud.
†Co-corresponding authors.

manager ensures the health and scaling of the job and task executors. DEEPFLOW deploys JEs and TEs dedicated to post-training and serving. We focus on serving in this paper.

Serving Engine. In DEEPFLOW, we design an efficient serving engine called FLOWSERVE, built on three fundamental principles. The first is a microkernel-inspired design, which divides system functionality into modular components that can scale independently. This separation ensures that different system parts can evolve and operate asynchronously. The second principle is NPU-centric execution, which means keeping the NPU busy to minimize delays caused by other resources such as CPUs, DRAM, or storage. Finally, FLOWSERVE adopts an Single-Program-Multiple-Data (SPMD)-based design, enabling efficient parallel processing and scaling across multiple NPUs.

The FLOWSERVE engine has six core functions: tokenizer, model execution, scheduler, memory management, caching management, and networking management. The tokenizer operates independently. Adhering to the SPMD design principle, FLOWSERVE follows a master-executor architecture: the master oversees scheduling, caching, networking decisions, while per-NPU executors carry out these decisions on their respective NPUs. FLOWSERVE's scheduler is centralized at the master module, using both asynchronous KV cache prefetch and asychnrous execution to keep the NPU busy. Additionally, FLOWSERVE builds a relational tensor cache (RTC) module to manage the relationship between tensors (primarily on the KV cache) and a distributed flow (DistFlow) module to transfer tensors across tiered storage within a single engine or across engines in a peer-to-peer manner.

Scheduling Algorithm. DEEPFLOW presents three designs to tackle the challenges introduced by prefix caching and disaggregation. First, we implement a locality-aware algorithm to maximize KV cache reuse, a design shared by [9, 32, 43]. Second, we extensively compare PD-disaggregated and PD-colocated TEs in a controlled environment with varying configurations. Based on the profiling results, we develop a PD-aware scheduling policy that accounts for the dynamics of online serving and decode length uncertainty. Finally, we propose a combined scheduling algorithm that integrates load-aware, locality-aware, and PD-aware strategies.

Scaling Optimization. DEEPFLOW achieves fast scaling by quickly adjusting to fluctuating workloads. It employs several key techniques, including reserving pre-warmed Pods and task executors (TEs), pre-loading models into DRAM, and leveraging high-speed NPU-to-NPU links for efficient model loading. Combined with optimizations like parallel initialization and predictive model pre-loading, these methods significantly reduce initialization time. DEEPFLOW can scale up to 64 instances in parallel within seconds.

This paper makes the following contributions:

- Design of DEEPFLOW, a large-scale serverless AI system for LLM serving (§3).
- Design of FLOWSERVE, an efficient and modular serving engine architecture (§4).
- Study of scheduling techniques for PD-disaggregated and PD-colocated setups (§5).
- Detailed description of optimization techniques for fast scaling in LLM serving (§6).

2 Background

We describe LLM and Ascend NPU chips in this section.

LLM. LLM generation has two main phases: the prefill phase and the decode phase. During the prefill phase, the model processes the full input prompt $(x_1, x_2, \ldots, x_n)$, computes the key and value vectors for each token, and stores them in the KV cache. A new token, x_{n+1}, is generated to kick off the decode phase. The prefill phase is compute-bound, leveraging significant parallel processing. In the decode phase, the model iteratively processes each new token. It computes and appends the corresponding key and value vectors to the KV cache, generating each subsequent token until the stopping condition is met. This phase is memory-bound, with the process being dominated by memory access rather than computation.

Ascend NPU. The Huawei Ascend Neural Processing Unit (**NPU**) is a special chip designed for AI models, built with the DaVinci architecture [20]. Each NPU chip has more than 20 DaVinci cores, which include independent cube cores and vector cores. Depending on the model, an NPU chip can deliver between 280 and 400 TFlops of performance for FP16 operations. Each chip also has 32 to 64 GB of high-speed memory (HBM), while lower models use LPDDR memory. Each server has eight NPU cards, connected through PCIe.

In an NPU cluster, two types of communication links are available for NPU-to-NPU communication. The first is a scale-up network (e.g., HCCS) that provides higher bandwidth and lower latency. The second is a scale-out network (e.g., RoCE) that supports larger-scale deployments, offering lower bandwidth but enabling the connection of thousands of nodes. To facilitate efficient communication across the NPU cluster, Huawei provides the Huawei Collective Communication Library (HCCL), which includes collective APIs like *all_reduce* and *broadcast*, as well as peer-to-peer APIs such as *send* and *recv*. The hardware trend is that the scale-up network domain is increasingly larger, enabling us to move from regular servers to so-called SuperPod and eventually fully disaggregated data centers [31].

3 DEEPFLOW: a Serverless AI Platform

DEEPFLOW is Huawei Cloud's fully-hosted and serverless platform for running emerging generative AI workloads, offering industry-standard fine-tuning, agent serving, and model

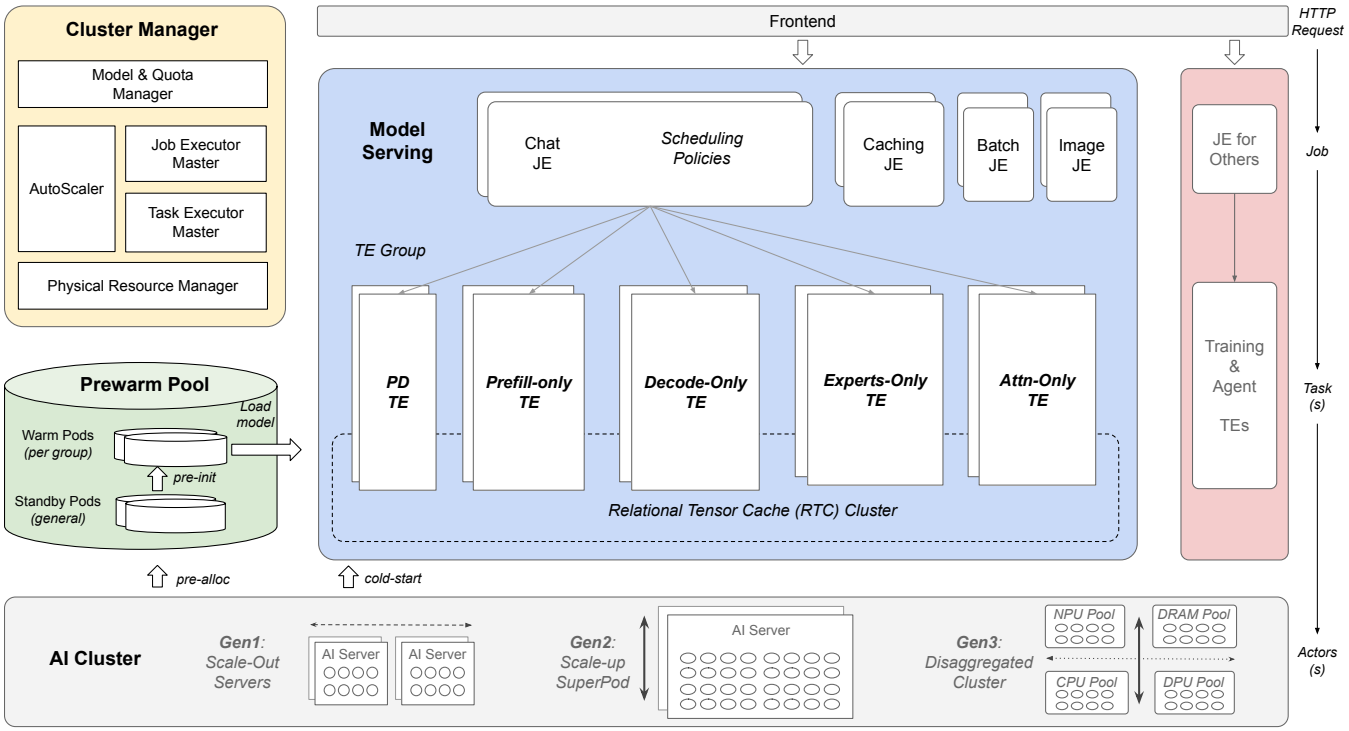

Figure 1: **DEEPFLOW Overall Architecture**. (a) User requests are routed through the frontend and dispatched to the appropriate JEs. (b) Model serving includes various types of JEs, such as those for chat completion and batch inference. (c) Each model-serving JE independently runs distributed scheduling algorithms. (d) Model-serving TEs in an RTC group can exchange tensors using DistFlow APIs (see §4.4). (e) The cluster manager is a centralized, high-availability module. (f) We omit post-training and agent serving for brevity. (g) We plan to run three generations of the NPU cluster. We now support Gen1 and Gen2.

serving APIs [25]. DEEPFLOW has been running for over a year atop a large Ascend NPU cluster.

As a cloud platform, our goal is to maximize AI cluster *performance* and *utilization* while ensuring SLO guarantees for multi-tenant users. However, achieving this with emerging AI workloads presents significant challenges. We have identified three key challenges.

- Challenge 1: AI workloads have varying durations, creating challenges for efficient resource sharing. Fine-tuning can take hours to days, agent serving ranges from seconds to minutes, and LLM serving typically lasts seconds to minutes. The challenge lies in dynamically allocating shared resources among these workloads without underutilizing or overloading the system.

- Challenge 2: LLM serving is becoming more distributed and stateful. A single inference request may span multiple distributed instances [10, 27, 44] and use cached states [9, 12, 43], making it challenging to efficiently manage resource allocation, synchronization, and fault tolerance across the system.

- Challenge 3: LLM serving is highly variable, leading to fluctuating resource demands [8, 41]. This introduces challenges in optimizing resource utilization and handling cold start latencies.

We build DEEPFLOW in response to these challenges. It is a scalable system that dynamically allocates and balances resources for post-training and serving workloads. As shown in Figure 1, DEEPFLOW has the following key designs.

Serverless Abstraction. DEEPFLOW introduces a unified execution abstraction to manage diverse post-training and serving workloads. This abstraction is built around three concepts: *request*, *job*, and *task*. Users interact with DEEPFLOW by sending HTTP requests, which trigger one or more internal jobs. Each job may generate multiple tasks. For example, a fine-tuning request triggers multiple internal jobs, including preprocessing, training, and evaluation; a chat completion request results in a single internal serving job. If executed on a PD-colocated engine [18], it creates one task; if executed in a prefill-decode-disaggregated setup [27], it creates two tasks; if executed in an attention-expert-disaggregated setup [26], it creates at least two tasks. This abstraction enables DEEP-

FLOW to scale AI workloads across shared infrastructure, share AI cluster among post-training and serving, and simplify the handling of distributed LLM serving.

Distributed Architecture. DEEPFLOW's architecture is built around its serverless abstraction and consists of three core components: job executors, task executors, and a cluster manager. The job executor (JE) processes incoming requests, decomposes them into jobs and tasks, and assigns tasks to available task executors for execution. The task executor (TE) is responsible for executing the tasks. DEEPFLOW deploys JEs and TEs dedicated to post-training, agent serving, and model serving. The cluster manager is a highly available system that oversees and scales all JEs and TEs. It includes centralized master modules for both JEs and TEs, each monitoring their health. Due to space constraints, this paper focuses on the model serving aspect.

Efficient LLM Serving. We design FLOWSERVE (§4), an efficient serving engine used by each model-serving TE. We deploy disaggregated serving optimizations for better SLO attainment. To support increasingly stateful and disaggregated serving, we propose distributed scheduling algorithms (§5) that run on model-serving JEs. Model-serving TEs running the same model and serving mode (e.g., prefill-only, decode-only) are organized into a TE group.

Fast Scaling. DEEPFLOW uses fast scaling to quickly adjust to changing online workloads. Key techniques include reserving pre-warmed Pods and TEs, pre-loading models into DRAM, and using high-speed NPU-to-NPU links for faster model loading. These methods, along with several optimizations, greatly reduce initialization time and allow DEEPFLOW to scale up to 64 instances in parallel within seconds.

In the rest of the paper, we will discuss **serving** in §4, **scheduling** in §5, and **scaling** in §6.

4 FLOWSERVE: an Efficient Serving System

This section presents FLOWSERVE, our in-house serving engine for large language models (LLM), large multimodal models (LMM), and embedding models.

4.1 Overview

When we started building FLOWSERVE in late 2023, we had three main goals. First, we wanted to create a high-throughput serving engine that works as a single TE. Second, we aimed for fast iteration, given serving is a fast-moving field. Finally, we wanted FLOWSERVE to be Ascend NPU-native, able to work with different generations of the Ascend cluster—from regular scale-out servers to SuperPod, and eventually fully disaggregated data centers [31]. To achieve these goals, we built FLOWSERVE based on three guiding principles:

- **Microkernel-inspired Design**: We apply the core principles of microkernel architecture by decoupling system functionality into modular, independently scalable components. This enables independent scaling, independent evolution, and asynchronous operations.
- **NPU-Centric Execution**: Our goal is to keep NPU busy at all times, reducing delays caused by waiting for other resources, such as CPU, DRAM, storage, or network.
- **SPMD-based Design**: We use the Single-Program-Multiple-Data (SPMD [4]) design to enable efficient parallel processing and scaling across multiple NPUs. This is similar to the recent vLLM v1 architecture [18].

Figure 2 presents FLOWSERVE' overall architecture. Each model-serving TE has a *TE-shell* and a FLOWSERVE engine. The *TE-shell* is an infrastructure module consisting of predefined modules such as scaling, health reporting, and a few customized modules that serve as context caching handlers.

The FLOWSERVE engine has six core functions: tokenizer, model execution, scheduler, memory management, caching management, and networking management. The tokenizer is an independent module that can scale on its own. Adhering to the SPMD design principle, FLOWSERVE follows a master-executor architecture: the master oversees scheduling, caching, networking decisions, while per-NPU executors carry out these decisions on their respective NPUs. We map the remaining functions, excluding the tokenizer, onto this master-executor design as follows:

- **Scheduling and Model Execution (§4.2):** The master scheduler assigns the next batch, and each executor runs the model's forward pass. The master broadcasts requests to all executors when initiating a batch.
- **Caching and Memory Management (§4.3):** We build Relational Tensor Cache (RTC) to unify the management both caching and memory allocation. The master maintains indexing structures like prefix trees [43], while each executor handles memory allocation.
- **Networking Management (§4.4):** We build Distributed Flow (DistFlow) to transfer tensors between model-serving TEs. It runs on each executor, providing memory-semantic transfer APIs for both the model generator and scheduler, supporting multiple backend transfers and Ascend cluster generations.

We implement FLOWSERVE primarily in Python, with RTC and DistFlow written in C++. While having design principles is important, translating them into an efficient implementation is a significant challenge. Figure 3 shows the decoding performance of FLOWSERVE across three versions over a three-month period. From v1 to v2, we introduced asynchronous scheduling and IPC optimization, which resulted in more than 2x improvements when the TPOT SLA was set to 50ms. From v2 to v3, our optimizations focused on data structures, sampling, and so on, which resulted in roughly 20% improvement.

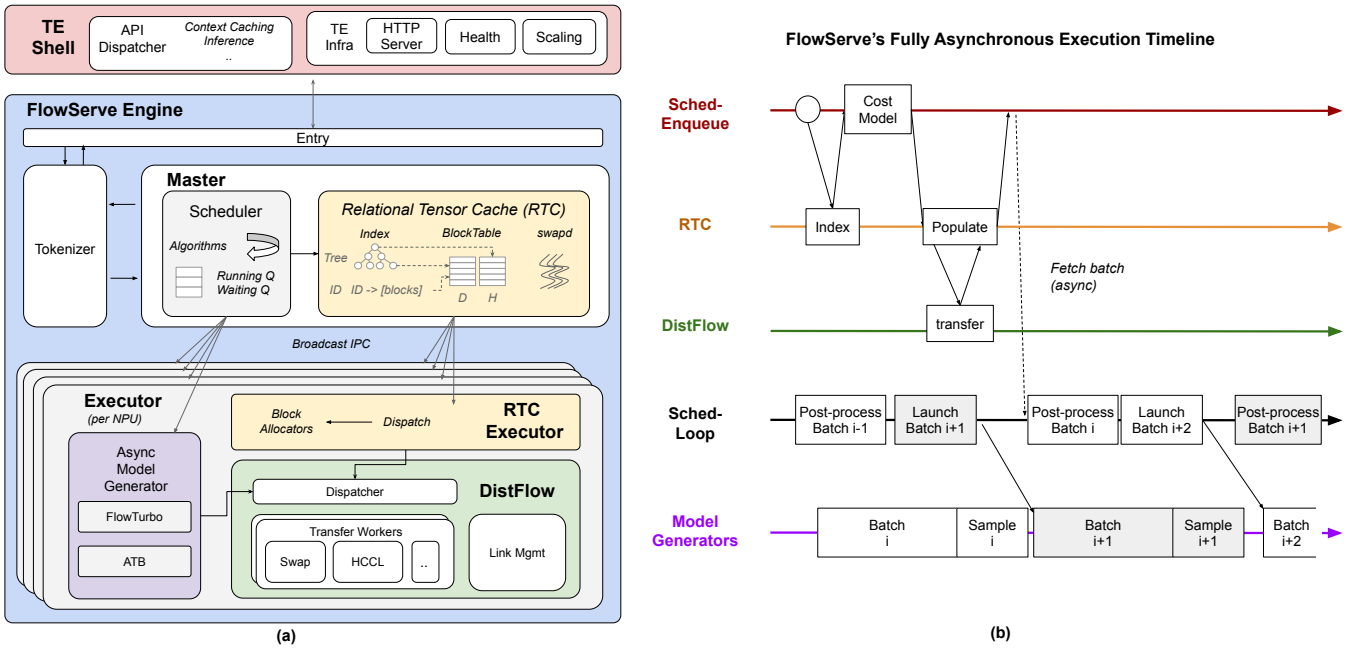

Figure 2: **FLOWSERVE Architecture and Asynchronous Execution Timeline.** (a) The FLOWSERVE engine can be deployed without the TE-shell. (b) The FLOWSERVE's model generator supports multiple backends. FlowTurbo is a torch-based dynamic graph. Ascend-Transformer-Boost (ATB) is a C++-based static graph [13]. (c) The FLOWSERVE engine is highly scalable, supporting deployments ranging from a single chip (TP=1) to clusters with tens of servers and hundreds of chips.

4.2 Scheduler

The scheduler is centralized in the master module. We prefer this approach over the distributed schedulers used by SGLang [43] due to its simplicity, albeit at the cost of frequent IPCs. The scheduler supports asynchronous KV cache prefetching enabled by RTC and DistFlow (described below) and asynchronous execution, a design shared by vLLM [18], SGLang [43], and NanoFlow [45]. We now describe the execution timeline in the right part of Figure 2.

Asynchronous KV Cache Prefetch. A sched-enqueue thread handles new tokenized requests from the tokenizer. When a request arrives, the thread follows these steps: First, the thread calls the RTC `match` API to check for any preserved KV cache. The returned info tells whether a prompt prefix or ID has cached data and their location (e.g., in the NPU or swapped to tiered storage). Second, if there is cached data but a portion of it is not in the NPU, the scheduler runs a fitted cost model to decide if reusing the cache is beneficial. If the cost model suggests reusing the cache can improve TTFT, the scheduler calls the `populate` API to request RTC to fetch the cache into the NPU. This step is done asynchronously. RTC will call DistFlow to read data from tiered storage or other TEs. Once RTC completes the cache population, it notifies the sched-enqueue thread, which marks the request as ready. The sched-loop will pick it up during the next scheduling cycle.

Asynchronous Execution. The asynchronous execution design aligns with our NPU-centric principle of fully utilizing the NPU. It is similar to the new zero-overhead scheduler introduced in SGLang-v0.4.0 [43]. The key observation behind this design is that scheduling decisions do not depend on the actual token IDs generated by the model, but rather on the number of tokens to be processed in each run. In typical decoding scenarios, where one token is generated per sequence per iteration, the scheduler can predict the required resources for the next run before the current one completes. This allows the scheduler to operate in a separate thread, preparing the necessary inputs for the model generator beforehand. By running the scheduler in parallel with the model execution, we eliminate unnecessary CPU wait times, ensuring that the NPU remains busy.

Pipeline Parallelism (PP). We optimize our scheduler for PP by running a centralized scheduler at the first stage of PP; other stages only accept requests from previous stages. This design enables FLOWSERVE to manage all micro-batches in a unified way. This approach has several benefits: (1) Memory resources are managed in one place, making it easy to preempt sequences across micro-batches; (2) With chunked prefill enabled, the scheduler distributes chunks across consecutive micro-batches, rather than sticking to just one micro-batch [1]. This helps reduce TTFT by at least 20%.

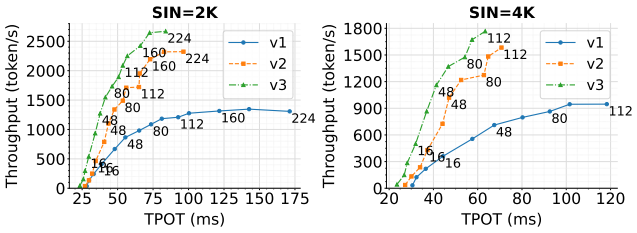

Figure 3: **FLOWSERVE Offline Serving Perf.** We run a 34B model with TP=4. The left has a prefill sequence length of 2K and the left is 4K. We run 256 decoding iterations and report the average TPOT and decoding throughput.

Table 1: The Core APIs of Relational Tensor Cache.

API	Description
`MatchByPrefixToken`	Find preserved KV cache by tokens
`MatchByID`	Find preserved KV cache by ID
`Populate`	Fetch preserved KV cache into NPU
`QueryPopulate`	Check populate status
`AllocBlocks`	Alloc blocks for prefill
`AppendBlock`	Alloc block for decode
`Copy`	Copy blocks from NPU to DRAM
`Free`	Free blocks

Data Parallelism (DP). We add DP attention to FLOWSERVE for DeepSeek-v3 models [21], inspired by SGLang's design [43]. At our scale, we address two key challenges. The first challenge is scalability. A single DeepSeek-v3 decode TE can require up to 20–40 servers. To scale FLOWSERVE across large clusters, we follow our microkernel-inspired modular design: we add a DP-scheduler after the tokenizer and create a separate per-DP master with its own scheduler and RTC modules. The DistFlow module, which scales independently, is shared across all DPs. The second challenge involves networking for mixed parallelism. DeepSeek-v3's attention and MoE parts can use different parallelism strategies due to varying AI chips and SLO requirements. For example, attention uses TP with DP, while MoE uses TP and EP, and the TP for attention differs from that of MoE. This requires using the right collective APIs to minimize network overhead and managing multiple communication domains as a result of it (e.g., hundreds of domains). These include one domain per DP attention group with a world size of attention TP, one domain per expert with a word size of MoE TP, and a single All2All domain for all experts.

4.3 Caching: Relational Tensor Cache

We build the Relational Tensor Cache (RTC) to integrate prefix caching [43] and position-independent caching [12, 37] into FLOWSERVE. The core function of RTC is to manage the *relationship* between tensors, primarily on the KV cache.

Abstraction. Table 1 lists the main APIs of RTC. RTC provides two main sets of APIs: The first set is for managing basic blocks, such as `AllocBlocks` and `AppendBlock`, which are used by prefill and decode requests, respectively. The second set is for managing KV caching. Notably, RTC includes various match APIs, each supporting different indexing methods, such as prefix-token-based or explicit ID-based matching. FLOWSERVE's implicit caching uses the prefix-token-based method, while the explicit ID-based method is used by DEEPFLOW's explicit context caching endpoint. RTC also offers a novel `populate` API. When invoked, RTC fetches the specified KV cache in the given range into local NPUs.

Design. RTC follows the master-executor architecture of FLOWSERVE. The master module manages the RTC module, making decisions about allocation and data movement, while each executor runs an RTC executor to perform these actions. Internally, RTC includes a traditional block table, originally proposed by vLLM [18], for managing data blocks. Additionally, RTC employs a hybrid indexing layer that combines radix-tree indexing [43] with ID-based indexing. In our current design, each index node can point to data stored either in the NPU or in local DRAM. We are also adapting RTC for SuperPod to leverage its global shared memory. Finally, the RTC master runs several background threads to handle tasks such as background swapping and prefetching.

4.4 Networking: DistFlow

We build the Distributed Flow (DistFlow) module to integrate advanced serving techniques, such as disaggregated prefill-and-decode [10, 27, 44] and disaggregated attention-and-experts [5, 26], into FLOWSERVE. The core function of DistFlow is to *transfer* tensors across tiered storage within a single TE and between distributed TEs in a peer-to-peer manner, which contrasts with collective-based networking functions used for tensor or pipeline parallelism.

Abstraction. DistFlow provides a set of control-plane APIs, such as `LinkCluster` for establishing connections, and a key data-plane API, `transfer(srcInfo, dstInfo)`. Users of DistFlow must provide the memory buffer addresses for both the source and destination; DistFlow does not operate with a block-based abstraction. In FLOWSERVE, the centralized scheduler, each executor's model generator, and the RTC executor can all invoke the transfer API.

Design. DistFlow follows design principles proposed by seminal data-center RPC libraries [15, 16, 24, 34]. The system most similar to DistFlow is the Mooncake Transfer Engine [28]. As shown in Figure 2, DistFlow supports multiple transfer backends. In a regular Ascend cluster, we primarily use HCCL peer-to-peer APIs, while in Ascend SuperPod, we adapt to standard memory copy primitives. Scalability is a key concern. To ensure this, DistFlow adopts a scalable threading model that avoids synchronization bottlenecks.

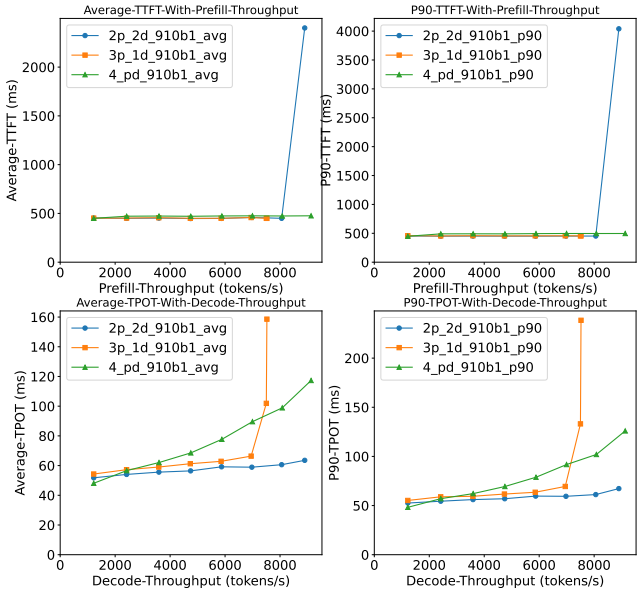

Figure 4: **FLOWSERVE Online Serving Perf.** We run a 34B model with TP=4 using an internal trace (roughly 2K input with 200 output). We test three setups: (1) PD-disaggregated with two prefill and two decode, (2) PD-disaggregated with two prefill and one decode, and (3) four PD-colocated. We vary RPS from 0.2 to 1.2 in a step of 0.2

4.5 Disaggregated Serving

Serving is increasingly disaggregated. In DEEPFLOW, we define two levels of disaggregation:

- **Task-level**. This refers to disaggregating prefill and decode into separate TEs. Our implementation is similar to [10, 27, 44]: FLOWSERVE in the prefill TE sends prefilled KV cache to decoding TE either by-req or by-layer using DistFlow's transfer API. Figure 4 shows an online serving test comparing PD-disaggregated and PD-colocated using an internal trace. Disaggregation greatly improves throughput under certain SLA and lowers TPOT with the same throughput.

- **Operator-level**. This refers to disaggregating attention and experts into separate TEs. We take inspiration from pioneering works in this space [5, 26]. We are actively working towards deploying it over SuperPod.

5 Distributed Scheduling

This section discusses DEEPFLOW's distributed LLM request scheduling policies.

5.1 Overview

In DEEPFLOW, distributed scheduling for LLM requests faces three new challenges. The first challenge involves locality and states. Before prefix caching was introduced into FLOWSERVE, distributed scheduling was straightforward, as all TEs could be treated as stateless, and scheduling was based solely on load. However, with the introduction of prefix caching, TEs have become stateful, and the goal is to schedule requests where KV cache reuse is possible, leading to the need for locality-aware scheduling. Achieving locality-aware scheduling is non-trivial. Selecting the best TE for a given request requires balancing KV cache reuse while avoiding overloading the TE that holds the most cache. The second challenge comes from disaggregated serving. Distributed scheduling was easier before PD-disaggregation was added to FLOWSERVE because all TEs were the same. However, with PD-disaggregation, choosing the best TE for a request becomes more complex, as it is unclear whether PD-disaggregated TEs or a PD-colocated TE will perform better. The last challenge arises from the coexistence of both prefix caching and disaggregation in DEEPFLOW, requiring any practical algorithm to consider both factors.

DEEPFLOW uses the following designs in response to these challenges: a locality-aware scheduling algorithm (§5.2), a PD-aware scheduling algorithm (§5.3), and a combined scheduling algorithm (§5.4) as listed in Algorithm 1.

5.2 Locality-aware Scheduling

In this section, we aim to answer one question:

> Given a request and a set of TEs with associated KV cache residency information, how can we select the TE that maximizes KV cache reuse to efficiently serve the request?

To address this question, we implement a prompt-tree-based, locality-aware scheduling policy in the `select_tes_prefix_match` function, as shown in Algorithm 1. Similar to Preble [32], SGLang [43], and MemServe [9], the distributed scheduler in JE maintains a global prompt tree for each type of TE, while each TE also maintains a local prompt tree that shares an index with its corresponding global tree. When a request arrives at the JE, it matches the prompt tokens against the global prompt trees and selects the TE that has the longest common prefix, indicating the largest preserved historical KV cache.

5.3 PD-aware Scheduling

In this section, we aim to answer one question:

> Given a request and a set of TEs with both PD-disaggregated and PD-colocated instances, how can we

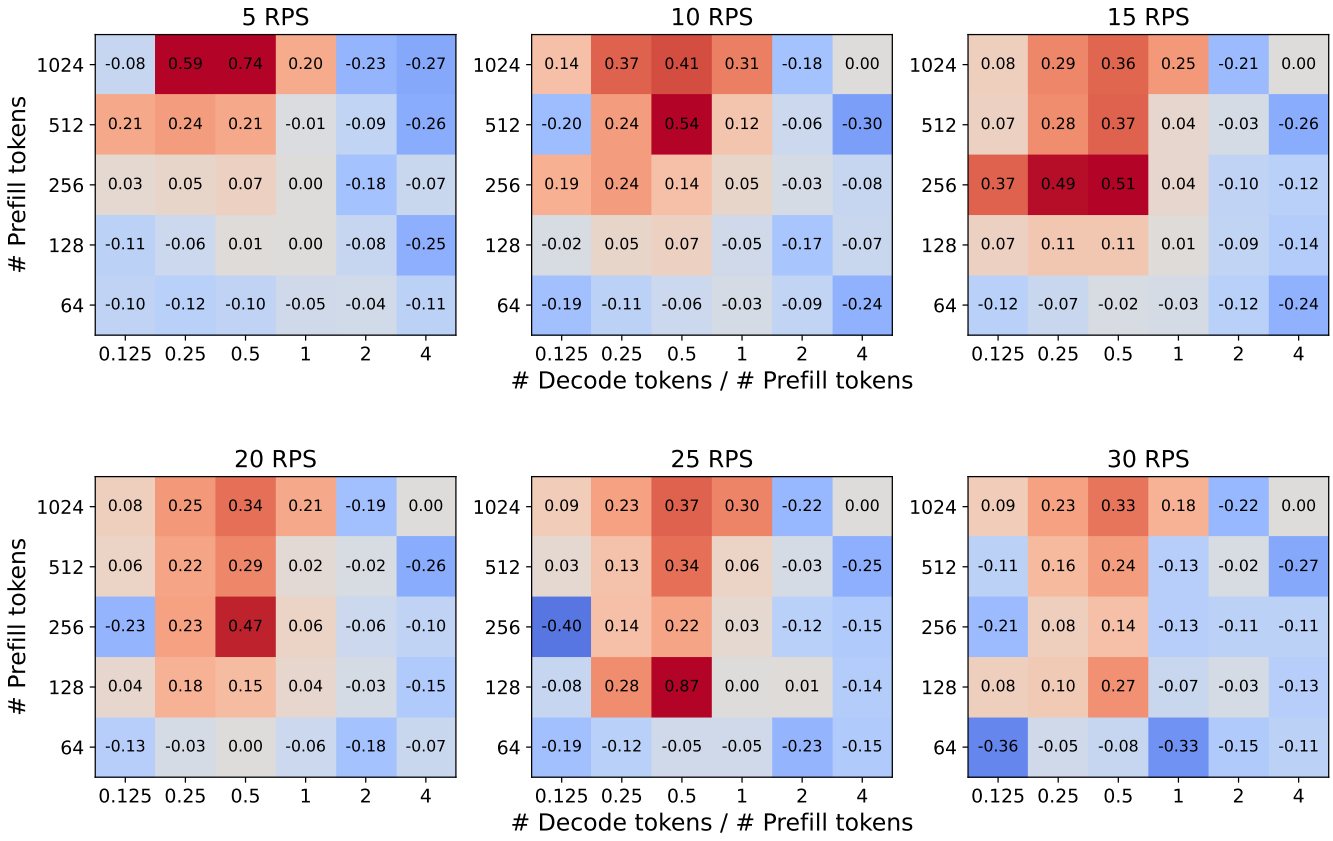

Figure 5: **Comparing the Performance of PD-disaggregated and PD-colocated (w/ chunked prefill) using Heatmap.** The y-axis represents the prefill length, and the x-axis shows the ratio of decode length to prefill length. For each combination of prefill and decode lengths, we execute a batch of identical requests at a fixed RPS on both PD-disaggregated and PD-colocated TEs. The heat map cells display the difference in JCT between the two setups, computed as the ratio of JCT for the PD-colocated TE to the PD-disaggregated TE, minus one. A positive value indicates better performance of the PD-disaggregated TE, while a negative value suggests the PD-colocated TE is more efficient. This figure uses a 34B model with TP=4.

select the most appropriate TE type that best aligns with the characteristics of the request?

We first study the performance comparison between PD-disaggregated and PD-colocated across various dimensions in a heatmap. We then discuss converting such a heatmap into a practical scheduling algorithm.

5.3.1 Study

We first run tests to compare PD-disaggregated and PD-colocated TEs (w/ chunked prefill). Figure 5 presents the results (see caption for detailed setup).

The heat map shows that PD-disaggregated and PD-colocated TEs divide the request space into distinct regions. We find three key observations. First, the PD-disaggregated setup performs better for requests with longer prompts and shorter decode. Additionally, as prefill length increases, its advantage becomes more pronounced for requests with longer decode lengths. Second, PD-disaggregated TEs provide a larger performance advantage over PD-colocated TEs (dark read) than the reverse (light blue), suggesting that a correct choice of PD-disaggregated TE leads to significant performance gains, while an incorrect choice results in minimal loss. Finally, the advantage of PD-disaggregated and PD-colocated TEs remains consistent across different RPS values.

5.3.2 Algorithm

The above study is conducted in a controlled environment, and applying it to a practical scheduling algorithm is challenging due to the dynamic nature of RPS and the uncertainty of decode length at the time of scheduling. There are several potential approaches to convert the study into an algorithm.

We propose a simple policy called `select-tes-PD-heatmap`, described in Algorithm 1. The policy works as follows: First, we combine the heat maps across all RPS values through element-wise addition. Given the stability of the heat map, over 80% of the squares consistently show either positive or negative values across all RPSs, while the remaining 20% are uncertain. Second, we predict the decode length for an incoming request using a set of decode length predictors with varying accuracy. These predictors are integrated into the scheduler to assess how prediction accuracy affects overall performance. One such predictor is the oracle, which assumes perfect accuracy and is an upper bound for performance. In practice, we use a predictor with 90% accuracy to balance prediction precision and overhead. Third, we identify the corresponding square on the heat map based on the combined heat map, the prefill length, and the predicted decode length. If the value is positive, we select PD-disaggregated TEs; if negative, we select PD-colocated TEs.

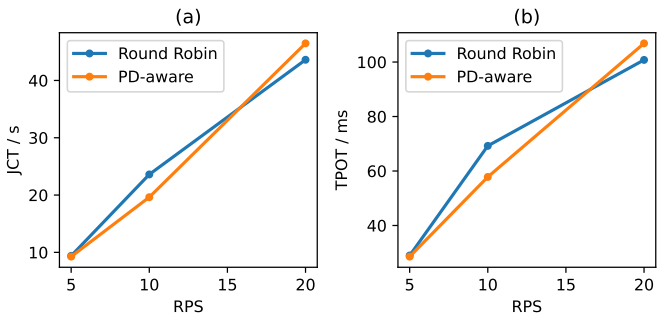

Figure 6: **Distributed Scheduling Algorithm Study.** We run a 34B model with TP=4, and report JCT / TPOT. We run an internal trace sampled from a code generation service. The cluster consists of four servers with two PD-colocated TEs and a pair of PD-disaggregated TEs (1P1D).

We perform a microbenchmark study in Figure 6. We make three key observations. First, under certain RPS levels (e.g., 10 reqs/s), the PD-aware scheduling policy outperforms RR. Second, at low RPS levels, the performance of PD-aware matches that of RR. This is because, under low RPS, interference between prefill and decode operations within PD-colocated TEs remains negligible, and PD-disaggregated TEs do not offer significant advantages. Third, at very high RPS levels, PD-aware performs worse than RR. This occurs because PD-disaggregated TEs, with the same resources (e.g., two cards), are more prone to overloading. However, even when overloaded, PD-aware scheduling does not exhibit significant performance degradation compared to RR.

5.4 Combined Algorithm

Algorithm 1: Distributed Scheduling Policy

Input: A new request: req, a TE group: tes
Output: A TE to forward the request to

```
Function dist_sched(req, tes):
    tes ← PD_aware (req, tes);
    if tes.is_load_balanced() then
        tes ← locality_aware (req, tes);
    else
        tes ← load_aware (req, tes);
    return tes;
Function PD_aware(req, tes):
    p_l ← req.get_prefill_length ();
    d_l ← req.get_decode_length ();
    tes ← tes.select_tes_PD_heatmap (p_l, d_l);
    return tes;
Function locality_aware(req, tes):
    tes ← tes.select_tes_prefix_match (req);
    return tes;
```

We propose a combined algorithm that integrates load-aware, locality-aware, and PD-aware scheduling. The scheduling algorithm, outlined in Algorithm 1, relies on three core function calls: `locality_aware`, `PD_aware`, and `load_aware`. These functions progressively refine the TE group, narrowing it down from the entire set to a single TE based on the request's characteristics and the underlying TEs.

The scheduling process proceeds as follows. First, a subgroup of TEs is selected by choosing a specific type, such as PD-colocated or PD-disaggregated TEs, based on the request's length and the heat map of the TE group (Section 5.3). Second, once a subgroup of TEs is identified, the selection is further refined. If the load is balanced across the remaining TEs, the algorithm prioritizes a TE with the most reusable KV cache, using tree-based prefix matching (Section 5.2). If the load is unbalanced, the algorithm instead prioritizes a TE with the least load to achieve better load distribution.

6 Fast Scaling

In DEEPFLOW, the cluster manager's AUTOSCALER determines when to scale TEs and JEs based on metrics like load or SLO-violation rates. Fast scaling is critical but particularly challenging for LLM serving TEs, as it requires loading large model weights onto NPUs. This section describes our optimizations for fast scaling.

Figure 7 shows the autoscaling workflow. Our mechanism involves five steps, with challenges and solutions summarized in Table 2 and performance breakdown in Figure 8.

ID	Step	Definition	Major Issues	Our Solutions
1	Scaler-Pre	Creating pods to hold the TE.	1. Resource allocation is slow	1. Pre-warmed Pods
2	TE-Pre-Load	Launching the TE w/o model loading	1. Python startup is slow 2. NPU init is slow	1. Pre-warmed TEs
3	TE-Load	Loading the model onto the NPU	1. Model weight is large	1. DRAM pre-loading 2. NPU-fork
4	TE-Post-Load	Preparing TE to serve requests	1. Engine warmup is slow 2. block alloc is slow	1. Offline profiling 2. Async allocation 3. Dummy req warmup
5	Scaler-Post	From TE ready to serve first request	1. TE list retrieval interval	1. Proactive pushing

Table 2: **A Summary of DEEPFLOW's End-to-End Scaling Steps, Challenges, and Solutions.**

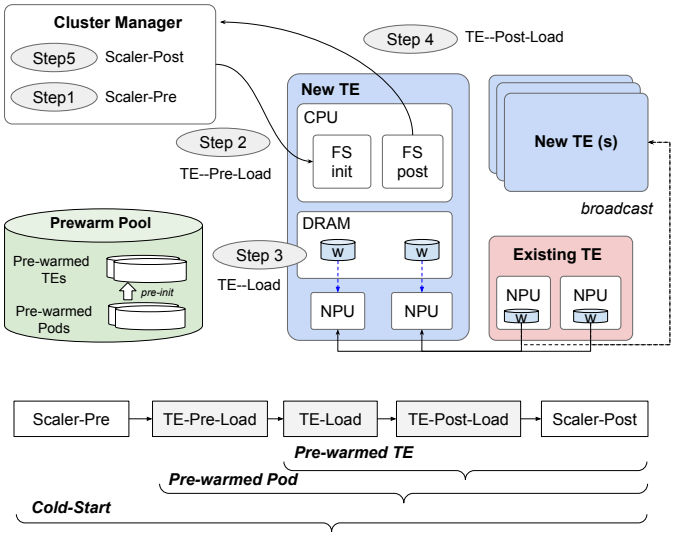

Figure 7: **DEEPFLOW's Scaling Design.** (a) We show TE-Load's two cases: loading from local DRAM (DRAM-hit) and loading from other TE's NPU (NPU-fork). NPU-fork can be either via scale-up or scale-out network link. (c) The bottom shows the timeline for three scaling cases.

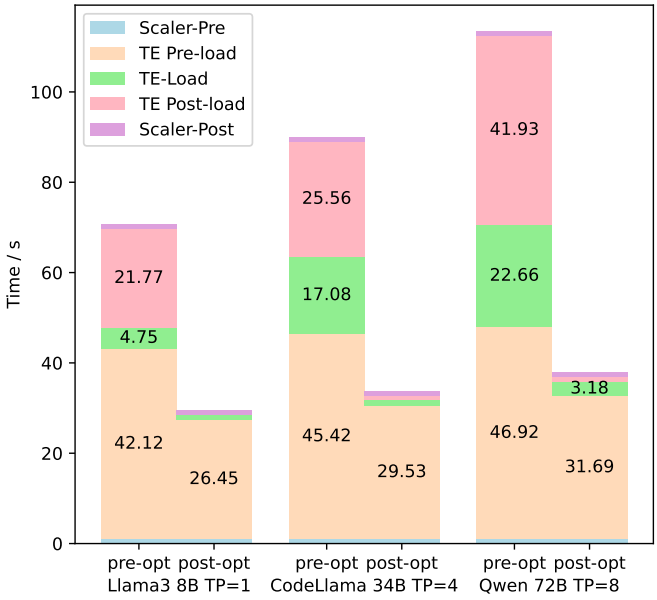

Figure 8: **Scaling E2E Breakdown**. We present the scaling latency both before and after optimizations. Even after optimization, the *TE-Pre-load* step remains the dominant factor in scaling time, though this can be further reduced through pre-warming (§6.1).

First, in the *Scaler-Pre* phase, DEEPFLOW prepares resources and pod environments for the new TEs. For large models, a TE may span multiple pods on multiple machines. Second, the *TE-Pre-Load* phase initializes the FLOWSERVE instance but does not load the model onto the NPU. Third, the *TE-Load* phase covers loading model weights onto the NPU. We decouple these two steps to allow for pre-warmed TEs, facilitating faster scaling (see §6.1).

The fourth step, *TE-Post-Load*, involves engine warmup and CPU/NPU block allocation. Open-source inference engines like vLLM [18] rely on warmup to profile the allocable HBM size for KV caches. We found this step unnecessary in production, as we offline profile this data and store it in configurations. To address the slowdown of the first request after removing warmup, we added a dummy message post-startup. Finally, in the *Scaler-Post* phase, the new TEs are announced to the cluster, and JEs direct requests to them. This wraps up the auto-scaling process.

6.1 Pre-warming Pods and TEs.

DEEPFLOW maintains two levels of pre-warmed resources. First, at the pod level, DEEPFLOW reserves a small number of pre-warmed pods with basic environment setup. These workload-independent pods are usually managed by the infras-

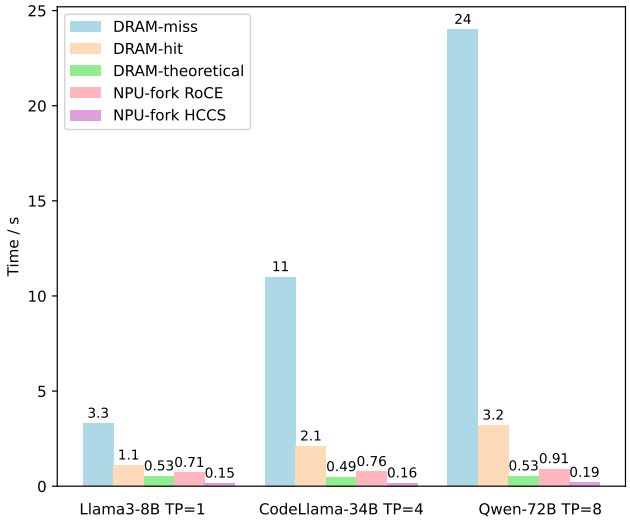

Figure 9: **TE-Load study.** DRAM-hit means loading from pre-loaded weights in DRAM; DRAM-miss means pre-load miss so loading from SSD; DRAM-theoretical is calcuated by model weights dividing PCIE bandwidth. NPU-fork has two different links, HCCS and RoCE (see text for the differences).

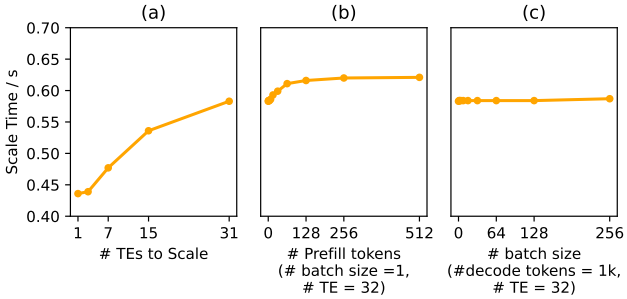

Figure 10: **Scalability and Sensitivity Study of NPU-fork**. We run NPU-fork over scale-up network (HCCS) using Llama3-8B-TP1. (a) Scaling multiple TEs in parallel from one running TE. (b) Time for scaling to 32 TEs when source TE is prefilling sequences of different lengths (c) Scaling time when the source TE is decoding different batches of sequences each with length of 1k tokens.

tructure layer, such as Kubernetes, and can be shared across services to reduce overhead.

Second, on these pre-warmed pods, DEEPFLOW maintains a small pool of pre-warmed TEs to minimize *TE-Pre-Load* time. The *TE-Pre-Load* step includes FLOWSERVE's startup time, which involves loading Python libraries, initializing NPU states, and setting up HCCL cross-NPU interconnections. We optimized this step by approximately 35% for most models using techniques like late importing and parallel initialization. However, as shown in Figure 8, this step still accounts for the majority of startup time. To address this, we move the initialization out of the critical path by incorporating a TE pre-warming mechanism.

Our TE pre-warming design was carried out in two stages. First, we made the pre-warmed TEs model-agnostic. For example, a TP-8 pre-warmed TE can be adapted to run either a Llama3-70B or a Qwen2-72B model. This required carefully distinguishing between model-specific and model-agnostic parameters. In the second stage, we made the pre-warmed TEs agnostic to parallelism strategies. This was achieved by recognizing that, regardless of TP/PP/SP configurations, all TEs follow a master-SPMD architecture. This allows independent pools of pre-warmed SPMD-masters and SPMD-executors, which can be packed on demand.

6.2 Optimizing Model Loading

DEEPFLOW has two model loading paths: *local loading* via PCIe from local DRAM or SSD, and *NPU-fork* using high-speed NPU-to-NPU links from a running TE. While NPU-fork is generally faster, it has higher hardware requirements and cannot be used during code start (scaling from 0 TEs).

Local loading with pre-warming. We use the safetensors format [30] for model storage. Compared to the native binary format, safetensors reduces serialization costs by storing tensors in contiguous blocks, which can be directly mapped into memory and only trigger data reads on page faults. This simplifies pre-loading into the page cache, reducing read amplification. Instead of loading the entire model file, each TP process loads only the required partition on demand.

To further optimize TE-load performance, we co-designed it with our TE pre-warming mechanism. The cluster manager predicts models likely to scale and pre-loads them into DRAM pagecache using pre-warmed TEs. When scaling is triggered, the master prioritizes selecting pre-warmed TEs with the required model already loaded. On our hardware, each machine has 1.5TB of DRAM, sufficient for pre-loading 10 70B models or 100 7B models.

Figure 9 shows TE-load's performance under different cases (NPU-fork, DRAM-hit, and DRAM-miss). DRAM-hit occurs when the cluster manager correctly predicts the models to scale, while DRAM-miss reflects the opposite scenario. We also computed the theoretical model loading time by dividing the model weights by the PCIe bandwidth. For DRAM-hit, the difference between measured and theoretical times arises from two factors: PyTorch model tensor initialization (Õ.3s) and PCIe bandwidth contention. The latter becomes more significant with larger TP ranks due to shared PCIe links. For the

three models, the weights loaded by each NPU are roughly the same, but local loading time increases with larger TP ranks due to PCIe link sharing among NPUs.

NPU-fork. NPU clusters typically feature high-speed NPU-to-NPU links designed for distributed training. In §4.3, we demonstrated how these links are used for efficient inference. We further exploit them in our NPU-fork technique to transfer model weights during fast scaling. When NPU-fork is triggered, the master notifies a running FLOWSERVE TE to connect to a pre-warmed TE, with model weights transferred using the DistFlow module (§4.4).

Figure 9 shows the performance of NPU-fork. On our Ascend hardware, there are two types of links: HCCS (higher bandwidth, smaller scale) and RoCE (lower bandwidth, scalable to thousands of nodes). All evaluations were conducted on cross-node setups. Overall, Loading with HCCS is significantly faster than with RoCE, indicating that NPU-fork will benefit from the superpod architecture. The model loading time is similar across the three models, as the weights to be loaded per NPU are roughly the same. NPU-fork experiences less bandwidth contention compared to local loading, as they use different physical links.

Figure 10 shows two sensitivity tests for NPU-fork. It can scale to a large number of TEs by transmitting model weights simultaneously to multiple TEs using the `broadcast` API in the HCCL collective communication library. We also measured resource contention sensitivity when the source TE handles prefill and decode requests. Since the NPU has dedicated AICPU for data transfer, contention is limited.

7 Related Work

In this section, we review related work in four key areas: serverless computing, serving engines, scheduling, and scaling for large language model (LLM) workloads.

Serverless. Extensive research has focused on optimizing ML model serving in serverless architectures [2, 6, 19, 29, 36, 39, 40], with industry solutions such as AWS SageMaker and Azure ML [23], and the open-source KServe [17], providing practical implementations. To the best of our knowledge, DEEPFLOW is the first publicly described serverless LLM serving system with detailed abstraction and internal design.

Serving. Serving is a rapidly evolving field, with many open-source engines like FLOWSERVE emerging. vLLM [18] pioneers PagedAttention for higher throughput, SGLang [43] uses RadixAttention for reusing cache, and LightLLM [11] adopts asynchronous execution for improved throughput. We believe that all serving engines will eventually adopt similar architectures to optimize AI chip efficiency, with differences mainly in hardware support, features, and language. Networking-wise, the Mooncake Transfer Engine (MTE) is closest to FLOWSERVE's DistFlow. Both systems support multiple backends and efficient threading models, following seminal works [15, 16, 34]. The key difference lies in backend implementation: MTE uses RDMA by default, while FLOWSERVE uses HCCL's peer-to-peer APIs.

Scheduling. Efficient scheduling is critical for improving the performance of serving systems. [1, 7, 9, 18, 27, 28, 35, 38, 43, 44]. For example, at the local layer, Orca [38] proposes iterative-level scheduling to reduce bubbles. Sarathi [1] proposes chunked-prefill to overcome suboptimal prefill processing. FastServe [35] utilizes a multi-level priority feedback queue to minimize JCT. At the global layer, MuxServe [7] formulates a multiplexing problem and proposes a placement algorithm and adaptive batch scheduling strategy to identify optimal colocations in LLM serving. MemServe [9] prioritizes locality by directing requests to instances with the highest cache hit rate. Our work introduces a novel PD-aware scheduling policy to determine whether a request should be processed by PD-disaggregated or PD-colocated TEs. Additionally, our approach integrates PD-awareness with locality- and load-aware scheduling, providing a comprehensive solution for optimizing resource utilization and throughput across heterogeneous TE configurations.

Scaling Optimizations. Scaling serving instances dynamically is a major challenge, mainly because these instances are large and continue growing as modern LLMs size increases. Recent work on model autoscaling [3, 14, 22, 42] has focused on improving resource use and reducing scaling delays. For example, SpotServe [22] and Llumnix [33] speed up scaling by making task migration between instances cheaper. BlitzScale [42] introduces a method that improves throughput while loading parameters, which helps reduce delays in handling burst requests and improves the overall service. Our NPU-fork mechanism is similar to BlitzScale [42]. To the best of our knowledge, this paper is the first to explain the complete scaling process, its challenges, and the optimizations involved.

8 Conclusion

This paper presents DEEPFLOW, a serverless AI platform developed at Huawei Cloud. We describe its serverless abstraction and infrastructure, which enable efficient management of AI workloads. The paper also outlines the architecture of FLOWSERVE, our in-house serving engine, detailing its design principles and core components. Additionally, we explore distributed scheduling across a heterogeneous pool of serving instances. Finally, we provide an end-to-end analysis of fast scaling, highlighting the techniques that allow DEEPFLOW to quickly adjust to fluctuating workloads.

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 69. IEEE/ACM, 2020.

[3] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 499–514. USENIX Association, 2020.

[4] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ML. *Proceedings of Machine Learning and Systems*, 2022.

[5] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. Efficient and economic large language model inference with attention offloading. *arXiv preprint arXiv:2405.01814*, 2024.

[6] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-gpu servers with spatio-temporal sharing. In Jiri Schindler and Noa Zilberman, editors, *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 199–216. USENIX Association, 2022.

[7] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple LLM serving. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[8] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[9] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024.

[10] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[11] Jiawei Hu, Hong Jia, Mahbub Hassan, Lina Yao, Brano Kusy, and Wen Hu. Lightllm: A versatile large language model for predictive light sensing. *arXiv preprint arXiv:2411.15211*, 2024.

[12] Junhao Hu, Wenrui Huang, Haoyi Wang, Weidong Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. Epic: Efficient position-independent context caching for serving large language models. *arXiv preprint arXiv:2410.15332*, 2024.

[13] Huawei. Mindie atb models.
https://www.hiascend.com/document/detail/zh/mindie/10RC2/mindiellm/llmdev/mindie_llm0004.html, 2025. Accessed: 2025-01-14.

[14] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 249–265. ACM, 2023.

[15] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[16] Anuj Kalia, Michael Kaminsky, and David G Andersen. {FaSST}: Fast, scalable and simple distributed transactions with {Two-Sided}({{{{{RDMA}}}}}) datagram {RPCs}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[17] KServe. KServe.
https://github.com/kserve/kserve.

[18] 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*, 2023.

[19] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In Jiri Schindler and Noa Zilberman, editors, *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 2022.

[20] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44. IEEE Computer Society, 2019.

[21] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[22] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 1112–1127. ACM, 2024.

[23] MinIO. MinIO.

https://min.io.

[24] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling rdma rpcs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[25] OpenAI. Openai, 2025. Accessed: 2025-01-14.

[26] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. Instinfer: In-storage attention offloading for cost-effective long-context llm inference. *arXiv preprint arXiv:2409.04992*, 2024.

[27] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[28] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[29] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In Irina Calciu and Geoff Kuenning, editors, *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 397–411. USENIX Association, 2021.

[30] The safetensors contributors. safetensors: A safe and efficient format for tensor serialization, 2023. Accessed: 2025-01-14.

[31] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2022.

[32] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving. 2024.

[33] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 173–191. USENIX Association, 2024.

[34] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[35] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[36] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 768–781. ACM, 2022.

[37] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving with cached knowledge fusion. *arXiv preprint arXiv:2405.16444*, 2024.

[38] 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)*, 2022.

[39] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 138–148. IEEE, 2021.

[40] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In Dahlia Malkhi and Dan Tsafrir, editors, *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1049–1062. USENIX Association, 2019.

[41] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Fast and live model auto scaling with o (1) host caching. *arXiv preprint arXiv:2412.17246*, 2024.

[42] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Fast and live model auto scaling with o (1) host caching. *arXiv preprint arXiv:2412.17246*, 2024.

[43] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. 2023.

[44] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[45] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.