DyOrc: Efficient Serving of Dynamic Machine Learning Workflows

Shiwei Zhang
The University of Hong Kong
u3007964@connect.hku.hk

Lansong Diao
Alibaba Group
lansong.dls@alibaba-inc.com

Zisheng Meng
The University of Hong Kong
u3012682@connect.hku.hk

Siyu Wang
Alibaba Group
siyu.wsy@alibaba-inc.com

Wei Lin
Alibaba Group
weilin.lw@alibaba-inc.com

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

Abstract

The landscape of machine learning applications has shifted from monolithic end-to-end models to compositions of pretrained large foundation models. For instance, multi-modal chatbots are often built by composition of a large language model and modality-specific encoder models. Such applications often feature dynamic workflows, with models conditionally evoked according to different inputs and intermediate processing results. Conditional model execution prevents conventional request batching and hinders efficient hardware utilization, due to dynamic, diverging execution paths across requests. Separately deploying models as dedicated services and invoking them on the go during dynamic workflow executions can potentially allow service-wise request batching, boosting resource efficiency. However, generic workflow orchestrators are proven inefficient for machine learning applications, due to schedulers that do not exploit batching, communication methods that are suboptimal for GPU tensors, and the considerable cold-start delays associated with model deployment.

We introduce DyOrc, an efficient serving system for dynamic machine learning workflows with conditionally activated model components. DyOrc deploys each model as a dedicated service and scales each service according to demand. Requests from different workflows that share the same component are batched on the go to improve hardware utilization and reduce serving latency. DyOrc further advocates three core designs: (i) speculative scheduling, that utilizes runtime statistics to predict component invocations for more informed inference scheduling and request batching among models; (ii) multi-tier message passing, which runs multiple services in one process and exploits specialized tensor communication libraries to substantially reduce communication overhead of heterogeneous messages across components; and (iii) proactive component loading, that cooperates with the speculative scheduler to hide cold-start overhead of components with billions of parameters. Extensive prototype experiments on representative inference workloads (including multi-modal chatbots, ensemble models, image processing pipelines, and multi-agent debate systems) demonstrate that DyOrc achieves 4-198% serving latency improvement over available baselines such as Ray Serve and Triton Inference Server.

CCS Concepts

• **Computing methodologies** → **Distributed computing methodologies**.

Keywords

Dynamic workflow, Workflow scheduling, Machine learning

ACM Reference Format:
Shiwei Zhang, Lansong Diao, Zisheng Meng, Siyu Wang, Wei Lin, and Chuan Wu. 2025. DyOrc: Efficient Serving of Dynamic Machine Learning Workflows. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772218

1 Introduction

On pursuing artificial general intelligence, the landscape of machine learning (ML) applications has evolved from end-to-end models trained to solve specific tasks (e.g., image classification, dialogue generation, speech recognition) to composition of foundation models that can handle different tasks on different modalities of input [27, 63]. A composite ML application can be created by connecting multiple pretrained models with lightweight adapters. For example, an image captioning application can be built by combining a ViT [28] model and a large language model (LLM), using a small adapter that aligns embeddings of text tokens and image patches. Graphical tools like ComfyUI [12], Langflow [14], and Coze [13] allow users to create applications by compositing popular models without writing code, fostering building a diverse array of applications on top of a small set of foundation models (e.g., image editing apps based on LLM and diffusion models [39]).

Serving these applications requires invoking the model components in certain order and transferring data across components. Such an application may contain tens or more components [62] (e.g., up to 26 LLM components in KNOWLEDGECARD [32] for text generation), forming complex directed acyclic graphs (DAG). In some applications, components are dynamically invoked according to modalities or contents of inputs, further complicating efficient serving [32, 42]. For example, the multi-modal chatbot GILL [42] invokes the stable diffusion model only for requests that involve image generation. The prevalent approach, such as the built-in inference script of GILL, is to load all components of the application onto a given set of devices and serve individual requests without batching, since different requests may invoke different components. This serving approach can cause significant underutilization of ML devices, which are typically designed for batch processing [19, 31, 43, 64].

We investigate more efficient component deployment and serving of composite machine learning applications with dynamic workflows. In our context, a serving system manages a collection of resources (e.g., GPUs) and provisions inference services for multiple ML applications. These applications may share certain components, such as a popular LLM. The application developer submits the inference workflows to the serving system and establishes APIs for end users. Upon receiving a user request, the application enqueues it in the serving system to schedule its inference and awaits results to respond to the end user. The serving system dynamically loads and runs model components according to incoming requests of applications, aiming to minimize end-to-end serving latencies of requests within limited resources.

Primary challenges in achieving such a serving system lie in dynamic deployment scaling of different components to execute the diverse workflows and efficient request batching for expedited inference serving. While previous research has extensively investigated batching multiple requests for *static* workflows [34, 66] and *intra-request* invocation batching during dynamic workflows [31] (i.e., batching multiple invocations to the same model layers in a single request), there remains a significant gap in *cross-request* batching under *dynamic* workflows. In applications with dynamic workflows, the execution of different requests may activate different components. For the example of multi-modal chatbot GILL, one request can entail generating an image response and another a text response, invoking different components such as StableDiffusion[54] and text decoding. On the other hand, requests originated from different applications may invoke the same component (e.g., an LLM), and can be temporarily combined into a batch for that component's processing – an opportunity rarely explored in existing multi-application serving systems. Moreover, under dynamic workflows, the optimal ratio of replications of each component should vary depending on the system's current workload [52].

Exploring these opportunities, we deploy each model component as an independently scalable service and advocate component-level dynamic request batching. Inference serving of composite applications is then structured as workflows that invoke different services at different times. Importantly, we batch invocations of each service regardless of different workflows the requests may belong, allowing dynamic batching across different applications. Although the concept is intuitive, we identify several challenges that hinder naive implementation from attaining ideal performance. *First*, commonly employed workload scheduling algorithms [7, 8] such as Round-robin and backpressure routing do not exploit batch processing capabilities of GPUs, resulting in substantial resource inefficiency. *Second*, large tensors are exchanged among components, incurring considerable overhead with standard serialization and communication methods [2, 3, 5]. *Further*, state-of-the-art models possess large numbers of parameters, incurring significant cold-start overhead for dynamic component deployment. Especially, existing workflow orchestration systems load services reactively [1], i.e., starting loading a service when it is being invoked, leading to latencies that far exceed user expectations.

Addressing these issues, we propose DyOrc, a serving system that facilitates efficient deployment and serving of novel composite machine learning applications with dynamic workflows. The key designs and core contributions are summarized as follows:

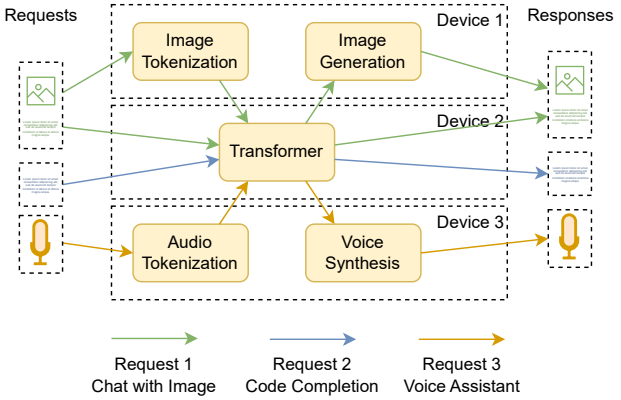

Figure 1: Three applications built on five components.

▹ We examine characteristics of modern composite ML applications and identify issues of their current serving methods, revealing the need for component-level dynamic batching. We then propose to deploy each component as an independent service to exploit dynamic batching opportunities across requests.

▹ We propose a speculative scheduling approach to efficiently schedule dynamic workflow execution, that minimizes the estimated job completion time based on a prediction of the services that may be invoked by the workflow.

▹ We design a multi-tier message passing scheme that cuts the serialization and communication overhead of large tensors on GPUs. We advocate a totipotent process design to enable direct GPU tensor sharing between services placed on the same device, and use NCCL [41] for accelerated tensor transfer among devices.

▹ Leveraging predictive information within our speculative scheduler, we enable proactive service loading and CPU caching, in advance of their actual invocation, mitigating the cold-start overhead.

▹ We build DyOrc on PyTorch and conduct extensive experiments on real-world datasets to compare it with various baselines for serving representative composite applications. DyOrc achieves up to 198% latency improvement compared to baselines.

2 Background

2.1 ML Applications with Dynamic Workflows

Boosted by recent advancements in large language models (LLMs), general-purpose foundation models have emerged (e.g., SEED-LLaMA [36], EMU2 [56], LLaVa [47]) that can serve as the building blocks for complex workflows to enable various novel applications. Fig. 1 illustrates a serving system with five model components that powers three applications. The chatbot application may take a mixture of texts and images as input and invoke image tokenization (a visual encoder such as ViT [28]), transformer, and image generation (a visual decoder such as unCLIP-SD [53, 54]) components for multi-modality generation. Various tasks can be performed with these applications, including image captioning, image editing and visual question answering, with different workflows. Requests are dynamically routed to their respective components according to different modalities of their inputs and output of the transformer: requests containing images are routed to the image tokenization model first

before the transformer, while others directly to the transformer; then requests on which the transformer generates special `[IMG]` tokens are routed to the image generation module, while the rest are directly returned to the user. The code completion application and voice assistant application share the transformer component with the chatbot application, and may use other components. Invocations of the transformer can be batched for better hardware utilization, even when the requests originate from different applications.

As a practical ML application with dynamic workflows, KNOWLEDGECARD [32] includes a set of smaller LLMs called knowledge cards that are specialized on various domains (e.g., movies, sports, literature) and a general-purpose LLM. For each user request, it first identifies the domain of the question, invoking the corresponding knowledge cards to retrieve related knowledge, and then employs the general-purpose LLM to synthesize the response. Multiagent Debate [29] includes multiple language model instances that propose and debate their individual responses for multiple rounds, to arrive at a common final answer.

Such composite applications are typically not built by training a composite model in an end-to-end manner, but by combining components pretrained on different tasks and datasets [29, 32, 42, 47, 48, 56]. This results in a discrepancy between training and inference workflows. For instance, GILL [42] trains its visual encoder with captioning loss on image-caption pairs that can be naturally batched together. During serving, the requests may contain different numbers of images, requiring different numbers of invocations of the visual encoder in their serving. Most existing studies on composite applications focus on training [32, 35, 36, 42, 56], whose techniques may not be directly applicable to efficient serving of these composite applications.

2.2 Model Inference Serving

A number of ML serving systems have emerged in recent years, e.g., Ray Serve [8], KServe [10], TensorFlow Serving [50] and Triton Inference Server [7]. At the core of these systems are inference engines (e.g., TVM [23], TensorRT [9], TensorFlow[17], or PyTorch [51]), which accept *batches* of requests and run the inference operations on accelerator devices like GPUs and TPUs. Built on top of inference engines, API endpoints in the serving systems receive and batch requests, provision and schedule inference operations, and respond the results to users.

These serving systems are typically designed to optimize the serving of a single model or a set of independent models. While some has built-in support for dynamic workflows (e.g., "ingress deployment" in Ray Serve and "business logic scripting" in Triton Inference Server), they mostly treat steps in a workflow as independent requests to the components, without dynamic workflow specific serving optimizations.

In addition, Teola [57] orchestrates and optimizes static workflows for accelerated serving of composite applications, especially by concurrently invoking operations that are not directly dependent on each other. AsyFunc [52] deploys composite applications on serverless systems and scales different components differently to optimize resource efficiency in cases of light and heavy loads. They focus on optimizing the serving of a single, static workflow, while DyOrc is designed to serve multiple dynamic workflows.

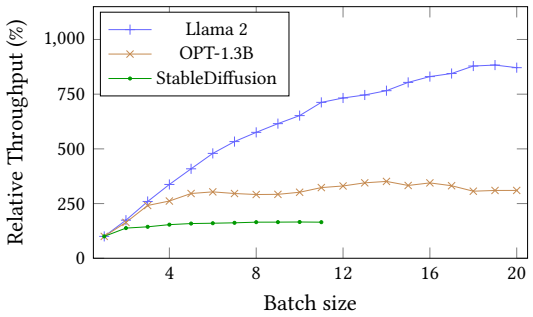

Figure 2: Throughput under different batch sizes (relative to batch size 1).

2.3 Workflow Orchestration

There have been workflow orchestration systems in data-intensive fields like data engineering and cloud computing [1, 4, 6, 15], managing complex data-processing workflows described by DAGs. They automate task execution, optimize resource use, and ensure reliable, scalable processing. These workflow orchestration systems focus on usability and reliability. Workflow management for costly ML serving should lean more towards the efficiency, i.e., hardware utilization and serving latency. For example, packaging components in Docker containers allows implementing them in different software stacks and programming languages, but also makes efficient message passing between components difficult, as containers are sandboxed and primarily rely on networking communication even if they are located on the same physical machine. On the other hand, if run on the same process, two components can directly pass data pointers and eliminate the serialization cost for messages, including tensors on GPUs.

3 Motivation and Challenges

3.1 Component-Level Batching

Batching is an important technique for efficient processing of large volumes of requests to machine learning models [24, 49, 66]. Inputs of requests in a batch are grouped into a high-dimensional tensor, and a single GPU kernel is launched to perform the same operation on all requests in the batch, utilizing parallel-processing capacities of accelerator devices.

There exists a well-known tradeoff between inference latency and throughput, when setting different batch sizes (i.e., the number of requests in a batch) [25]. As illustrated in Fig. 2, batched inference achieves 8x throughput on Llama 2 [59] as compared to running inference with batch size 1 (i.e., no batching), on an NVIDIA A100 GPU. Larger batch sizes allow higher serving throughput, while smaller batch sizes ensure shorter end-to-end inference delay for individual requests. On the other hand, in an online serving platform with limited computation resources, a throughput lower than the request arrival rate leads to queue build-up and request drops. It is necessary to dynamically configure the batch size to minimize inference latency while ensuring a throughput no lower than the request arrival rate.

Request batching is challenging in serving systems with dynamic workflows. If we use global batches of incoming requests as in

traditional serving systems, the control flow of requests in a batch may diverge during execution, and the requests need to be routed to different components. This essentially splits each batch dynamically into smaller batches according to the components the requests are routed to, causing the batch size to diminish.

For effective request batching under dynamic control flows, ACRoBat [31] leverages the structure of models such as RNN and TreeLSTM, and batch the invocations of the same model layer within a single request. Especially, it identifies invocations of the same model layer in a single inference request that do not have data dependencies, and run them in batches by concatenating the respective tensors in a "batch size" dimension. ORCA [64] exploits the autoregressive nature of LLMs to dynamically batch the decoding steps across different requests. LLM inference includes a prefill step which processes the initial prompt, and a series of decoding steps which autoregressively generate a new token based on previous tokens. The number of decoding steps of a request is dynamically determined during generation, typically by checking if a special `[EOS]` token is generated. These designs exploit characteristics of specific models such as RNN and LLM, and provide a crucial insight that can generalize to other dynamic workflows: by breaking down a dynamic workflow (e.g., LLM inference) into a series of steps (e.g., individual decoding steps), we can perform batching at a finer granularity and remove the dynamic control flow from the inference engine. The inference engine then performs batched inference on static fine-grained components, while a scheduler handles the dynamism in the workflow.

Proposed solution: Component-level Batching. Motivated by this insight, we advocate component-level batching in dynamic workflow serving of composite applications. In DyOrc, a composite application consists of several service functions, with each service function providing inference service of a specific model component. Requests can dynamically invoke different components in an application. Each component batches requests it receives for processing, with strategically set batch sizes.

3.2 Dynamic Component Deployment

The dynamic workflows challenge effective resource scheduling: how to optimally and dynamically place components onto given resources (e.g., GPUs) and schedule their serving to minimize the end-to-end inference latency of requests, without prior knowledge of components that will be invoked by each request. Mature workflow scheduling heuristics, such as HEFT [58] and PEFT [20], have been widely used for scheduling static workflows. For dynamic workflows, existing workflow orchestration systems often fall back to simple heuristics such as task-level greedy scheduling, which schedules each request serving task (a task is an invocation of a service) greedily onto a device with the shortest queue of this task. This approach can result in suboptimal performance due to high communication overhead between components in model serving. For example, suppose we have a chain-structured workflow A->B->C, where the invocation of task C is conditioning on the output of task B. Device D1 has loaded services A and B, while device D2 has loaded services B and C. If the scheduler is oblivious about a future task on C, it is likely to schedule a task on B on D1 to eliminate communication from A to B, if both devices are available at the time. This schedule can be suboptimal if the communication overhead of B->C is larger than A->B. A better schedule would be scheduling the task on B on D2 to remove the larger communication B->C, if we know the upcoming task on C. Therefore, a prediction of the complete workflow is desirable for efficient scheduling.

Proposed Solution: Speculative Scheduling. We propose to use the Markov process [30] to model the execution of workflows for future component invocation prediction, due to its simplicity and prediction speed (the scheduling overhead directly affects the performance of the online inference service). During runtime, DyOrc collects statistics of the dynamic workflow of each application, including the frequency of services that its requests invoke. With these statistics, DyOrc builds a speculative scheduler, which predicts the complete workflow of a request and derives optimal scheduling of the predicted workflow using dynamic programming.

3.3 Messaging Overhead

Different types of messages are communicated among components in a composite application. In serving a vision-language chatbot application, the messages can contain images, texts, or intermediate tensors. The common approach is to serialize the messages and transfer them over PCIe links or the network, when the components are deployed on different devices. However, this approach severely underperforms dedicated tensor communication methods, e.g., using NCCL [41], which may utilize dedicated hardware such as NVLinks to reduce communication latency.

Fig. 3 demonstrates a time breakdown of transferring 32 tensors of the shape (128, 768) between two GPUs, resembling the output of a text embedding layer with batch size, token length, and embedding size 32, 128, and 768, respectively. We use Python's `pickle` library for serialization, whose serialization and deserialization time includes the time spent on transferring tensors between GPU and CPU. When the two GPUs are on the same machine, the serialization and deserialization costs are up to 77% of the total communication cost. When the two GPUs are on two machines with a slow inter-connection, the serialization cost is still significant.

Existing generic workflow orchestration systems often run each service on dedicated processes or even in containers [1, 4, 6, 15]. In these cases, message serialization is needed even when the source and destination services are on the same device, due to not sharing the same address space. Ray Serve [8] adopts serialization-free and zero-copy communication of `numpy` arrays on the same CPU using shared memory. To our best knowledge, no widely used workflow orchestration system supports zero-copy communication of tensors on GPUs.

Proposed Solution: Multi-tier Message Passing. DyOrc advocates a multi-tier message passing scheme that specializes the communication methods based on the types and locations of the messages. For a message where the source and destination services are on the same physical device, DyOrc enables zero-copy message passing. With its totipotent process design (Sec. 6.1), DyOrc can directly pass pointers to the data, as the services are run in the same process and thus share the same address space and CUDA context, to skip serialization. For tensors transferred across GPUs, DyOrc utilizes NCCL [41] for accelerated transfer, potentially exploiting dedicated hardware such as NVLink.

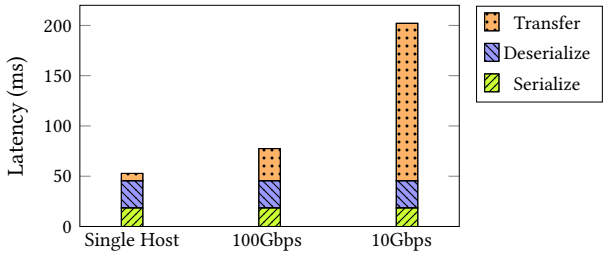

Figure 3: Communication cost breakdown.

3.4 Cold-Start Large Components

Contemporary machine learning models feature large numbers of parameters, whose retrieval from disk or remote storage (such as Amazon S3) consumes substantial time, prior to the initiation of services (i.e., the cold start problem [34, 40]). For example, loading the LLaMA 2 (7B) [59] model from hard disk takes 62 seconds, using Hugging Face Transformers [61] library. On the other hand, retaining these sizable components in GPU memory may waste valuable GPU resources, if they remain unused for prolonged periods. For KNOWLEDGECARD [32] which encompasses 26 LLMs tailored for various domains, a less popular domain receives infrequent requests, and reserving a GPU cache slot for rarely used LLMs does not justify the cost.

Proposed Solution: Proactive Service Loading. DyOrc endeavors to preload services prior to their immediate necessity. Specifically, DyOrc leverages information from the speculative scheduler to predict future service invocations and seeks to load the necessary services when a (predicted) invocation of that service is assigned to a device. If the available memory on the device becomes insufficient, DyOrc offloads the least recently invoked service to host memory, creating space on device for the new service.

4 DyOrc

We present DyOrc, an efficient serving system for composite machine learning applications. We assume a given cluster of GPU servers for serving multiple composite ML applications. The primary objective is to minimize end-to-end inference latencies of requests within the given resources.

Our design is guided by two key characteristics of inference workloads on composite machine learning applications. (1) Dynamic workflows: inference serving of an application is structured as workflows and requests to the same application may invoke different components for different times according to the inputs. (2) Component sharing: different applications may share the same model components. Dynamic workflows challenge classical global request-level batching, while component sharing opens a new opportunity for cross-request batching. Inspired by microservice architectures and serverless systems [33], we deploy each component as an independent service (not directly tied to any application), dynamically provision components to serve dynamic workflows, and advocate dynamic component-level request batching across applications.

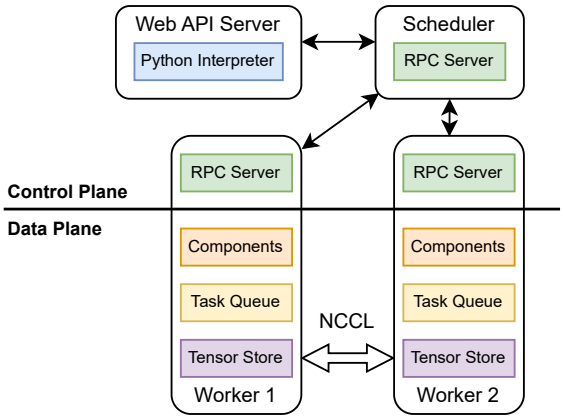

Figure 4: Architecture of DyOrc

```python
import dyorc

@dyorc.service
class ViT:
    def __init__(self):
        ... # load model checkpoint

    def __call__(self, *inputs):
        ... # perform inference with input batch

@dyorc.workflow
async def chatbot(img, txt):
    img_tokens = ViT(img)
    txt_tokens = Tokenizer(txt)
    outputs = LLM(img_tokens, txt_tokens)
    return await outputs
```

Listing 1: DyOrc API

Architecture. An overview of DyOrc is given in Fig. 4. On the control plane, the system includes a Web API server for receiving requests from users of the applications and a scheduler to schedule inference serving of the requests. The scheduler decides using which components deployed on which devices to process each request according to its dynamic workflow, referred to as the placement of each request's tasks (each task is an invocation of a service), and dispatches them to the respective devices. Each device (GPU) runs one worker process. DyOrc uses remote procedure calls (RPC) for the communication of the control messages. On the data plane, model components are deployed on workers. Each worker maintains an LRU cache of loaded components, a queue of scheduled tasks, and a tensor store that temporarily saves outputs of tasks. The workers transfer tensors to others using efficient communication libraries such as NCCL.

System API. DyOrc provides easy-to-use APIs for application providers to efficiently deploy their composite applications on our serving system. Listing 1 illustrates the key APIs, with an example

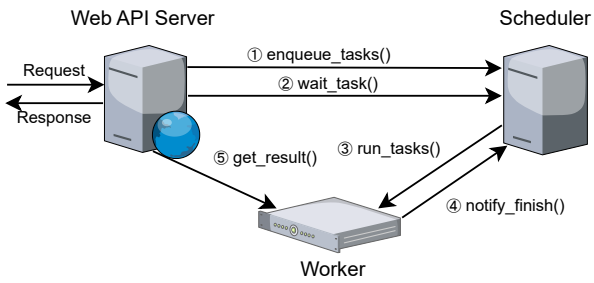

Figure 5: Working of DyOrc

of building a multi-modal chatbot. The application provider registers its components and workflows using Python's decorators syntax. Specifically, `@dyorc.service` is used to register a service in DyOrc and `@dyorc.workflow` registers a workflow, which dynamically invokes the services with given inputs. After registration, the Web API server can call the workflow function and `await` for inference results. Inside the workflow function, a call to a service enqueues it to the scheduler and returns a handler to the output data that can be passed to other services without actually retrieving the data. This avoids unnecessary round-trip transfers of potentially large messages between the Web server and workers, and allows concurrent invocations of services. When the output data are actually needed, the workflow function can `await` on the handler to retrieve the results.

Working of DyOrc is further illustrated in Fig. 5. Upon receiving a request of an application, the Web API server executes the `@dyorc.workflow` function and invokes `enqueue_tasks` on the scheduler via RPC. It also calls `wait_task` with the handler that the `@dyorc.workflow` function is awaiting. The scheduler subsequently invokes `run_tasks` on the designated worker (suppose only one worker is involved in processing the request), which later calls `notify_finish` to indicate the completion of the tasks. The scheduler communicates the worker ID back to the Web server by returning from the `wait_task` call. Finally, the Web server retrieves the output result from the worker using `get_result`. When the tasks for serving the request are scheduled to multiple workers and intermediate output communication is needed among the workers, the consumer of intermediate data uses the same `wait_task` to get the worker ID of the producer and then issues `get_result` to obtain the result, in the same way as the Web server does.

5 Speculative Scheduler

To schedule inference serving of an incoming request, the scheduler first utilizes the gathered runtime statistics and a Markov model to predict the future tasks that may be invoked by this request (Sec. 5.2). Then it executes a dynamic programming-based scheduling algorithm to determine a schedule for all tasks in the request (Sec. 5.3). Following this, the scheduler disseminates the schedule to the relevant workers and coordinates inference execution. When a worker process receives a task, it adds the task to its FIFO task queue, assigns it to the last batch if feasible (Sec. 5.1), and pre-loads the corresponding services if needed (Sec. 5.4).

5.1 Batching

DyOrc advocates batched task execution at the service level, enabling batching of tasks across various requests from different applications. Specifically, each worker combines multiple tasks of the same service in their queueing order in its task queue into a batch, up to a maximum batch size. DyOrc provides a heuristic to automatically determine the maximum batch size at each service, by profiling the service performance on a device with increasing batch sizes until the throughput stops increasing or out-of-memory error is encountered.

Input tensors of tasks within a batch are concatenated along the "batchsize" dimension and subsequently fed into the service. Upon completion, the batch is disassembled, and the output tensors corresponding to different requests are routed to their respective next-step services. A batch is formed immediately when the previous batch has been processed by the worker, so that the batch size may not always reach the maximum batch size but depends on how many tasks the scheduler has assigned to the worker by that time.

5.2 Workflow Prediction

During runtime, DyOrc collects workflow statistics of processed requests in each application into a frequency table, with rows in the form $(a, T_k) \rightarrow \{(S_0, f_0), (S_1, f_1), \ldots, (S_n, f_n)\}$. For each application a with tasks $T_1, T_2, \ldots, T_k$, f_i is the frequency that the next task invokes service S_i. Here $S_1, S_2, \ldots, S_n$ are registered services and S_0 indicates the end of the request.

The scheduler employs a Markov model to predict future tasks given a sequence of observed tasks in a request's serving. For an incoming request, a few initial tasks are often known, e.g., a request may first call a modality encoder, which serves as the observed task for predicting later tasks. Based on the initial tasks and the frequency table, the model autoregressively samples the next task, until S_0 is sampled.

5.3 Scheduling Algorithm

The scheduler decides which worker/device to dispatch each task of a request to, which consequently determines the components to load in the memory of each device and task batching at each service on each device. To minimize the end-to-end inference latencies of requests, the schedule should balance the following aspects.

• **Batch Sizes.** As discussed in Sec. 3.1, there is a trade-off in setting the batch sizes. The scheduler should strategically assign tasks of the same service to devices, to form batches of suitable sizes to the service.

• **Data Locality.** When inputs of a task t are on a device, assigning t to that device can eliminate communication and potentially allow t to start as soon as its inputs are produced.

• **Load balance.** The scheduler should prioritize assigning tasks to devices with shorter task queues; otherwise, some devices can be overloaded, resulting in significant queuing latencies.

• **Cold-start overhead.** When a task is assigned to a worker that has not loaded the respective service, it must wait for the worker to load the service first, before its execution. This cold-start time can be substantial due to the considerable number of parameters in some components.

• **Scheduling overhead.** As requests must wait while the scheduler is making the schedules, the scheduling algorithm should be fast to minimize the extra delay.

The scheduler schedules requests in a FIFO manner. Given the (predicted) complete workflow of each request, a speculative scheduling algorithm is designed that uses dynamic programming to determine the optimal device assignment of tasks within the request, aiming to minimize the end-to-end serving latency of the request. During actual request serving, if there is a discrepancy between the actual tasks and predicted tasks, request serving execution is halted and the dynamic programming algorithm is applied again with the updated workflow to identify an updated schedule.

The dynamic programming algorithm is based on the key structure that the earliest finish time of a task t depends on the placement and completion time of its input tasks and its own execution time on assigned device. We define a *partial schedule* P to be a set of triples (t, d, h), indicating that the output of task t is available on device d at time h. Starting from an empty partial schedule P_0, we keep deriving new partial schedules from a given partial schedule P following execution order of the workflow: for a task t whose input tasks are all scheduled in P, we enumerate device d and consider putting t on d. If any inputs are not on d, they are transferred to d before t starts. The finish time h of task t on d can then be calculated as follows:

$$h = \max\{\text{avail}(t, d),\ \text{ready}(t, d)\} + \text{comp}(t, d) \quad (1)$$

$$\text{ready}(t, d) = \max_{t' \in \text{inputs}(t)} \min_{(t', d', h') \in P} \{h' + \text{comm}(t', t, d', d)\} \quad (2)$$

Here, $\text{avail}(t, d)$ is the time when device d becomes available for task t, i.e., it has finished previously scheduled batches and loaded the service invoked by t, if not already cached in its memory. $\text{comp}(t, d)$ is the execution time of task t on device d, measured from the execution start to the completion of the batch that includes t on the device. $\text{ready}(t, d)$ denotes the time when the inputs for t are ready on device d, i.e., when outputs from all t's precedent tasks t' have been transferred from their respective devices d' to device d. For each input task t', we find a device d' which contains the output of t', from which the time to obtain the output to d is the shortest. a' is the time when the output is available on d', and $\text{comm}(t', t, d', d)$ is the communication time for transferring the output from d' to d. When $d' = d$, the communication time is considered 0, thanks to DyOrc's zero-copy communication design (Sec. 6.2).

Then we can add the triple (t, d, h) to the partial schedule P, resulting in a new partial schedule P'. This procedure repeats until all complete schedules are enumerated, after which we can identify the optimal schedule P^* that achieves the earliest finish time for the last task in the request.

As an important optimization to reduce the solution space, we remove redundant triples in any partial schedule P, and only keep partial schedules that are not outperformed by other schedules. A triple (t, d, h) is considered redundant in P if all tasks t'' that depend on t have been scheduled in P. A partial schedule P is outperformed by another partial schedule P' if for all $(t, d, h) \in P$, there exists $(t, d, h') \in P'$ such that $h' \leq h$. Algorithm 1 gives the detailed procedure. As the number of tasks within a request is limited, the optimal schedule for a single request can be attained without incurring significant overhead.

Algorithm 1 Dynamic programming algorithm for optimal workflow scheduling of each request

Input: Request r, devices $\mathbb{D}$.
Output: Optimal schedule P^*.

Initialize $\mathbb{P} = \{P_0\}$.
while $\mathbb{P}$ changed in last iteration **do**
  for $P \in \mathbb{P}$, $t \in r$ **do**
    if t not scheduled and all inputs(t) scheduled in P **then**
      for $d \in \mathbb{D}$ **do**
        Calculate h based on Eq. 1 and Eq. 2
        $P' = P \cup \{(t, d, h)\}$
        Remove redundant triples from P'
        $\mathbb{P} = \mathbb{P} \cup \{P'\}$
        Remove outperformed schedules from $\mathbb{P}$
      end for
    end if
  end for
end while
$P^* = \arg\min_{P \in \mathbb{P},\ P \text{ is complete}} \max_{(t,d,h) \in P} h$

The computation of $\text{avail}(t, d)$ accounts for component loading status on device d. If the service needed by task t is not loaded on d, $\text{avail}(t, d)$ could be large due to the need to load the service. Then the scheduler may favor other devices where the service is already loaded, unless the task queues of those devices are significantly longer. $\text{avail}(t, d)$ also takes batching into account: if there are other tasks on device d that invoke the same service and the maximum batch size is not reached, t can be added to the batch; otherwise, t has to wait for the current batch on device d to finish, resulting in a larger $\text{avail}(t, d)$ that may disfavour its placement on the device. When a device is overloaded, the algorithm tends to allocate tasks to other devices, even if this implies cold-starting a new instance of the service. Afterwards, new tasks invoking this service can utilize the newly loaded instance, alleviating the load on the overloaded devices. In addition, prediction of future tasks in a workflow allows the scheduler to assign tasks to devices where their consumers have also been loaded, reducing cross-device communication.

5.4 Proactive Service Loading

DyOrc enables two distinctive opportunities for alleviating the component cold-start problem. *First*, our speculative scheduler predicts and proactively schedules future tasks of requests, and services needed by later tasks can be pre-loaded. *Second*, the loading of model components on each worker can occur in parallel with the execution of other components. This is feasible because our component loading primarily involves CPU to GPU memory copy, and modern GPUs are equipped with dedicated copy engines that operate asynchronously alongside computation cores. Specifically, when the scheduler assigns a task to a device, the device endeavors to load the service function as early as possible, in parallel with inference execution. If the device's memory becomes full, it evicts the least recently used services that are not used for batches in the current task queue. Due to the predictive nature of our scheduler, DyOrc may engage in superfluous pre-loading by loading a service

for a predicted task that ultimately does not materialize. Nevertheless, given the considerable overhead associated with loading large components, we find this aggressive pre-loading strategy justified.

To further mitigate the cold-start time of services from hard disk or remote story, DyOrc supports caching components in host memory. Host memory is typically abundant in machine learning clusters compared to GPU memory. Upon initiating a new machine instance, DyOrc pre-loads all components involved in our serving system from disk or remote storage into host memory, significantly decreasing future component loading time.

6 Data Plane Design

6.1 Totipotent Worker Process

Conventional workflow orchestration systems [1, 4, 6, 15] launch dedicated processes to run instances of services. The processes can be placed on the same device, and are scheduled by the operating system. This dedicated process design can lead to high tensor communication overhead between processes, due to inter-process communication (IPC) which commonly involves serializing messages (with methods like FlatBuffers [2]) before transmission, even between two processes on the same device [3, 5].

We propose to run a single worker process per device. This worker process can execute any tasks by loading corresponding services. When deploying a service to a device, we simply instruct the process to load the corresponding component. Importantly, the services run in the same address space and CUDA context, allowing tensor data to be passed between them via passing data pointers and eliminating the need for any serialization. We call it a *totipotent process*, a metaphor for totipotent cells that can develop into any type of cell in biology. The worker process maintains a FIFO queue of tasks (interleaved with tasks to different components). A worker keeps checking the queue, executes the first batch of tasks from the queue, and loads the corresponding component if needed. The services loaded in the device memory are invoked in order of the tasks in the FIFO queue. The worker offloads the least recently used component when the GPU memory becomes insufficient.

The components are multiplexed on one device in a time-shared manner: the worker process only executes one batch of tasks of the same component at a time. This avoids resource contention between multiple running components while still allowing executing different components on the same device without reloading the parameters.

6.2 Multi-tier Message Passing

We design a multi-tier message passing scheme to accelerate communication between services.

Tier 1: Zero-Copy Communication. Enabled by our totipotent process design (Sec. 6.1), zero-copy communication is used for message transfer between services located on the same device. Since the services share the same address space and CUDA context, the messages can be directly passed by pointer, eliminating any serialization and network transfer cost.

Tier 2: Tensor Communication. In machine learning applications, most of the components communicate tensors (i.e., intermediate computation results) among each other. Tensors are highly structured, usually allocated on continuous memory buffers, and typically accommodated on GPUs. To transfer a tensor, generic communication methods employed by current workflow orchestration systems may first copy the tensor from GPU to host memory, wrap in a serialization format, then transfer over network. This method involves multiple copies of the potentially large tensor and cannot exploit dedicated GPU-to-GPU links such as NVLink. DyOrc detects the type of messages and automatically utilizes NCCL [41] for communication between devices if the messages contain tensors.

Tier 3: General Communication. A machine learning application may pass other kinds of messages, such as JPEG-encoded images, text strings, etc., as input and output. To maximize flexibility in implementing the components and applications, we serialize non-tensor messages and transfer them over the network using RPC, across devices.

Data Lifecycle. Each worker maintains a tensor store that retains the inputs and outputs of tasks for active requests (i.e., requests that have started but are not yet finished). When a task is assigned to a worker, the worker examines the tensor store for the required input of the task. If the input is absent, the worker queries the scheduler to determine the location of the task responsible for generating the input and subsequently retrieves the input data. Depending on the location and type of the input, the actual retrieval procedure may utilize one of the three tiers of message-passing mechanisms.

Upon completion of a batch, its outputs are saved in the data store, awaiting retrieval by other workers or utilization by subsequent batches on the same device. Given the dynamic nature of the workflows, DyOrc does not remove data from the tensor stores until the entire request has been served. We index the data using their respective request identifiers in the tensor stores and remove all data related to a request once the request is finished.

7 System Implementation

To serve model components as independent services, we advocate asynchronous Python function as the workflow definition, which provides maximum flexibility by allowing all kinds of control flows. Well-organized model code can be easily integrated into DyOrc by simply adding a few decorators (Sec. 4).

General workflow orchestration systems typically utilize language-agnostic formats, such as JSON, for workflow definition. However, since most composite models are implemented in Python, we integrate DyOrc with Python interpreters to alleviate the efforts for supporting these models. This also allows the application provider to insert custom logic between invocations of two components, enabling actions such as streaming intermediate output to the end user while the workflow continues to execute.

We implement DyOrc with about 3100 lines of Rust code for the scheduler and the worker and 360 lines of Python code for the API. DyOrc is a complete serving system that can execute user-defined workflows and service functions to serve requests for various composite applications. DyOrc supports workflows and components implemented in PyTorch, but can be extended to other inference engines with Python interfaces. DyOrc uses tarpc [16] for control plane messaging and both tarpc and NCCL for data

plane messaging. Python's `pickle` package is employed to serialize non-tensor messages for cross-device communication.

8 Evaluation

Testbed. We evaluate DyOrc on 16 machines in a public cloud with 128 GPUs in total. Each server is equipped with 8 NVIDIA A800 GPUs and NVLink.

Benchmark Models and Datasets. We evaluate DyOrc on three representative applications: (1) **GILL** [42] is a vision-language model that supports understanding and generating multiple images in a chatbot setting. We disable the retrieval ability of the model and only use its generation method. We use the VisDial [26] dataset as the inputs in requests to this model. (2) **KNOWLEDGECARD** [32] is an ensemble model that utilizes multiple domain-specific LLMs to support a larger LLM to generate more informed responses. To illustrate the unique strength of DyOrc, we use the "top-down" approach in KNOWLEDGECARD, which dynamically calls domain-specific LLMs and features a more complex workflow. We use the MMLU [38] dataset as input. (3) **DEBATE** [29] is a multi-agent system where multiple instances of LLMs debates for several rounds to answer a question. To demonstrate DyOrc's ability in handling dynamic workflows, we stop the debate as soon as a consensus is reached among agents (i.e., all agents give the same answer) at the end of a round, instead of running the debate for a fixed number of rounds as in the original implementation. We use the MMLU [38] dataset as inputs.

Furthermore, we evaluate DyOrc and the baselines by jointly serving the three applications, labeled as **MIX**. We share the same LLM component for KNOWLEDGECARD and DEBATE.

Workload. We evaluate DyOrc on a publicly available real-world trace [55] that has been adopted by related studies [22, 34, 44]. Following the practice in these studies, we map the functions in the trace to workflows in a round-robin manner and scale the trace to the desired request per second (RPS).

Baselines. We compare DyOrc with two state-of-the-art baselines. (1) **Ray Serve** [8] is a serving system that supports custom workflows built on top of Ray. We decomposite the applications and deploy each component as an independent actor in the same way of DyOrc. We use Ray Serve's default scheduler and built-in auto batching support. (2) **Triton Inference Server (TIS)** [7] is a multi-backend serving system. We use its Python backend to align the components and application implementations with DyOrc and Ray Serve. In particular, we implement the benchmark workflows using its Business Logic Scripting support, which allows running custom Python code on the serve that invokes other services.

TIS does not natively support automatic service loading and unloading, depending instead on manual service placement. For our evaluation, we first execute DyOrc in each scenario and then utilize the service placement chosen by DyOrc for the longest duration to deploy TIS. It is important to note that this approach is generally impractical, as the placement determined by DyOrc is contingent on the actual demands within the request traces. Consequently, the results are biased in favor of TIS.

8.1 End-to-end Serving Latencies

We serve each application for a duration of two hours with DyOrc and baselines. Figure 6 shows the average request latencies, which are measured as the total workflow duration, encompassing the time from request submission to final response retrieval. DyOrc demonstrates consistent and low latency across all applications, exhibiting only minor increases at the beginning of serving due to the inevitable cold-start cost of the modules. Overall, DyOrc achieves 4-198% average speed-up as compared to the best baseline for each application.

Ray Serve demonstrates competitive latency for the majority of the time in the KNOWLEDGECARD and MIX scenarios, but suffers from occasional latency spikes. These spikes can be partly attributed to Ray Serve's reactive service loading mechanism and the substantial cold-start cost associated with the machine learning services. As illustrated in Figure 7, the latency spikes exhibit a strong correlation with loading activities in Ray Serve. Owing to its reactive nature, Ray Serve initiates the loading of new component instances only when the task queue is already long. Consequently, these requests must endure the prolonged loading period of these components, resulting in significant delays.

In contrast, DyOrc proactively loads components that are predicted to be used. While this proactive approach may lead to superfluous loads, as indicated by the high loading frequency in Figure 7, DyOrc loads these components in parallel with ongoing computations and does not block the execution. As a result, the benefits derived from these proactive loads significantly outweigh the associated costs. TIS lacks the functionality to automatically load and unload services, necessitating manual configuration of each service's placement. This poses a significant challenge in real-world applications, as the optimal placement depends on incoming requests and may evolve over time. In our experiment, we employ an "oracle" placement for TIS, which is derived by DyOrc during serving the actual request trace.

We further demonstrate that the more frequent service loading does not impact the performance of DyOrc by presenting a breakdown of device times in Figure 9. We monitor the status of each device and categorize each moment into four states: (1) Running, when the device is processing a batch of service invocations; (2) Loading, when the device is loading a service; (3) Both, when the device is simultaneously running and loading; and (4) Idle. Our observations indicate that the majority of the loading time overlaps with computation, thus not reduce the overall device utilization.

The KNOWLEDGECARD model incorporates a large language model and a collection of domain-specific small language models. These domain-specific models are accessed only when required and are typically fast to load due to their smaller size. Therefore, DyOrc opts to load and unload the majority of these small language models on an as-needed basis, leading to a higher loading frequency. The DEBATE model, conversely, comprises only one component. Both DyOrc and Ray Serve refrain from unnecessary loading and unloading of this component. However, DyOrc loads the component onto all devices more promptly, resulting in a smaller latency spike at the beginning. In contrast, Ray Serve loads additional instances after a delay, leading to higher latency spikes.

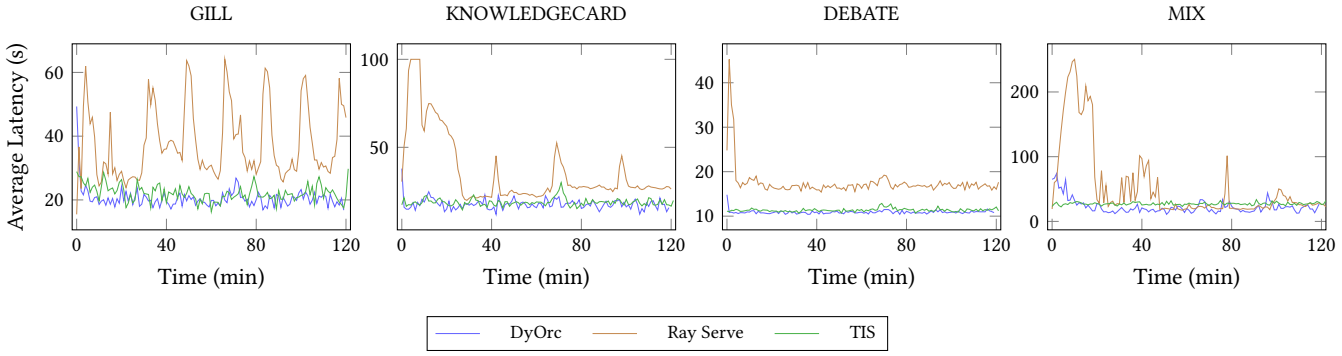

Figure 6: Average end-to-end request latency.

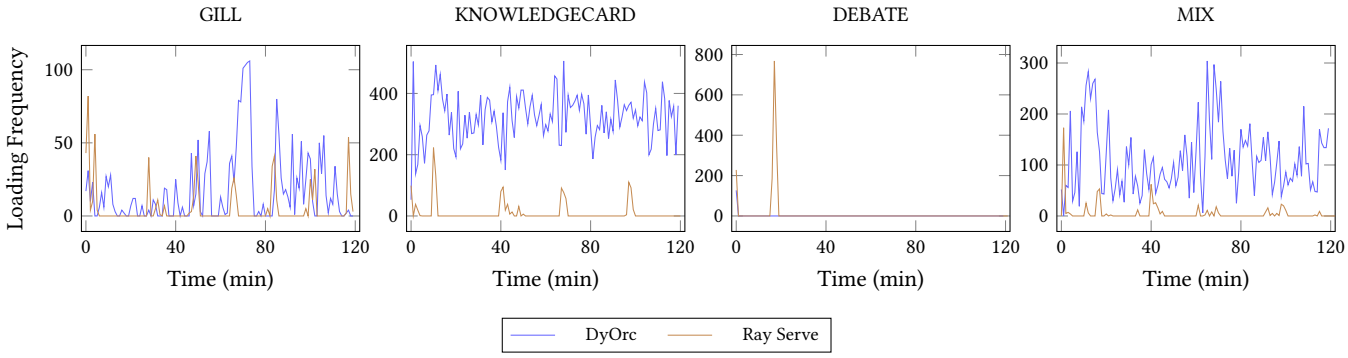

Figure 7: Service loading frequency.

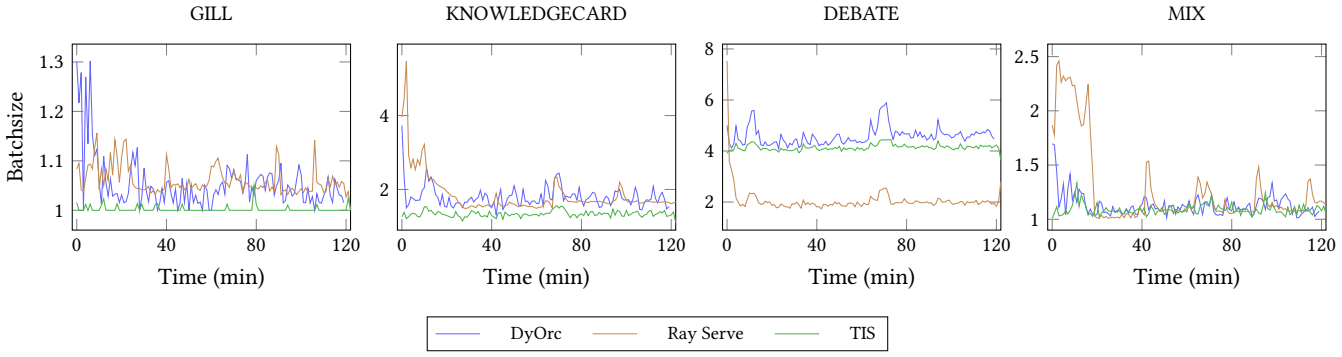

Figure 8: Average batchsizes.

The DEBATE case demonstrates another advantage of DyOrc's totipotent worker design for serving multiple components on one device. Ray Serve supports deploying multiple services on the same GPU by setting the resource requirement to fractional numbers. However, the services deployed on the same device run concurrently, and the scheduling of their execution is managed by the operating system and driver software. This often results in interleaved execution of several services at the CUDA kernel level, leading to unnecessary delays. When serving DEBATE, Ray Serve loads and executes multiple instances of the LLM on each device, causing reduced batch sizes (Figure 8) and longer overall latency. Setting the resource requirement to 1 for all services, effectively dedicating each device to a single service, allows Ray Serve to achieve comparable performance to DyOrc and TIS in the DEBATE scenario. However, this configuration leads to significantly poorer performance in other applications, as many devices will remain idle when their loaded service is not invoked, and the reloading frequency will significantly increase, exacerbating the cold-start issue.

In DyOrc, each worker may load multiple services if memory permits; however, only one service can be actively running at any

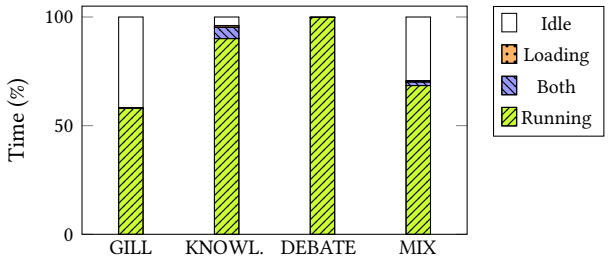

Figure 9: DyOrc device time breakdown. *Both* denotes the time when the device is simultaneously loading services and running batches.

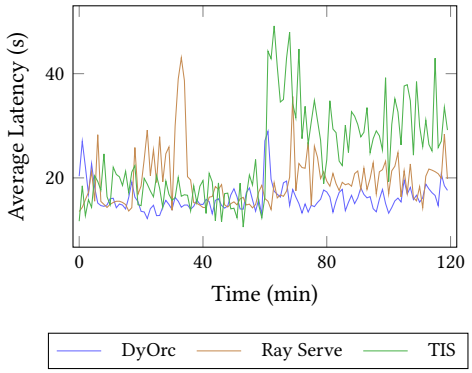

Figure 10: Average latency serving MIX. The request pattern is changed at the 60th minute.

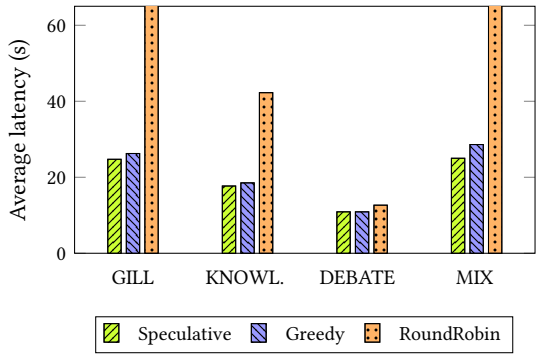

Figure 11: Latency of different scheduling algorithms.

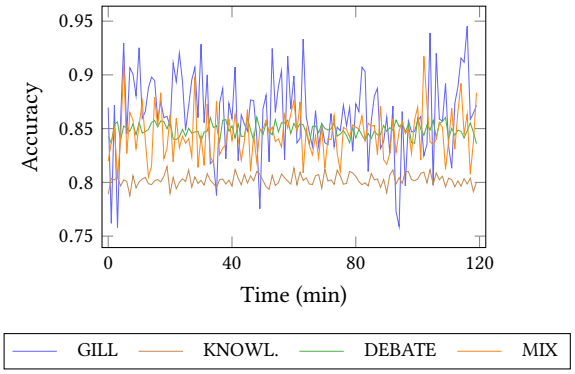

Figure 12: Prediction accuracy of Markov model.

given time. This design enables each worker to switch between the multiple loaded services to process requests without incurring the cold-start cost, while simultaneously ensuring efficiency by preventing the concurrent execution of multiple services.

Figure 8 illustrates the average batch sizes employed during the serving of the applications. All systems exhibit a tendency to utilize larger batch sizes under high server load, such as at the beginning of serving, while opting for smaller batch sizes during periods of lighter load to reduce latency. This observation corroborates our intuition presented in Sec. 3.1.

8.2 Serving Changing Requests

We evaluate DyOrc's ability to adapt to changing volumes of requests to different applications by synthesizing a trace that alters request patterns. Specifically, when serving MIX, we double the requests to GILL while halving the requests to the other two applications at the 60-minute mark. As shown in Fig. 10, the performance of TIS degrades significantly when the request pattern changes due to its static placement of components. Ideally, more instances of GILL should be deployed after the change to accommodate the increased number of requests, while the number of KNOWLEDGECARD and DEBATE replicas should be reduced accordingly. DyOrc and Ray Serve maintain similar latency after the change because they automatically scale the number of replicas of the components to meet the demands of the requests.

8.3 Speculative Scheduler

We evaluate the speculative scheduler of DyOrc by comparing it with two alternatives: (1) **Greedy**, which employs task-level greedy scheduling by assigning each individual service invocation to the device with the earliest estimated finish time, and (2) **RoundRobin**, which assigns each task to devices in a round-robin fashion. We conduct this experiment on a single machine in our testbed. Figure 11 shows that RoundRobin performs poorly in all cases except for DEBATE, because it assigns different tasks to devices without considering the services already loaded on those devices, leading to an excessive amount of service loading (the DEBATE model has only one component and thus does not require additional loading). The speculative scheduler exhibits similar performance to Greedy in KNOWLEDGECARD and DEBATE, and outperforms Greedy in GILL and MIX with improvements of up to 14.5%. In KNOWLEDGECARD, the selection of the next component to invoke is based on the prompt content, which is challenging to predict using our Markov process. DEBATE does not necessitate service loading, thereby reducing the advantages offered by our speculative scheduling.

8.4 Prediction Accuracy

We evaluate our Markov model for workflow prediction, as its performance is crucial to the speculative scheduler. As shown in

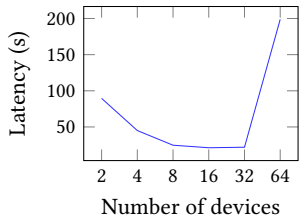

Figure 13: Average latency when serving GILL.

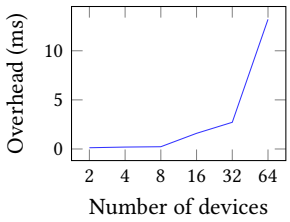

Figure 14: Scheduling overhead.

Fig. 12, the accuracy of the Markov model ranges from 76% to 94%. The accuracy heavily depends on the nature of the workflow. For example, the KNOWLEDGECARD workflow invokes different small language models ("knowledge cards") based on the topics of the prompt, making it difficult to predict since the prompt is not part of the input to our Markov model.

The scheduler predicts future tasks within a workflow before making scheduling decisions, which introduces a tradeoff between prediction accuracy and overhead. While employing a more powerful machine learning model could yield higher prediction accuracy [67], it would also increase scheduling overhead. Given that DyOrc utilizes a centralized scheduler design, this overhead is a critical concern. Therefore, we adopt a simple Markov model to achieve a balance between accuracy and overhead.

8.5 Scaling Performance

There are two methods for deploying DyOrc on large clusters. In the first approach, all devices are directly managed by a single DyOrc instance. However, this can lead to the scheduler becoming a bottleneck. The alternative approach is to partition the cluster into several subclusters, each managed by a separate instance of DyOrc, and to employ an external load balancer (e.g., Nginx at the Web API gateway) to distribute requests across these subclusters.

We experiment with DyOrc at varying scales with the first approach. As depicted in Figure 13 for the GILL workflow, DyOrc initially demonstrates a continuous decrease in latency as more devices are added. This is attributed to the limited number of loaded services that can be accommodated when the cluster size is small, necessitating repeated loading and unloading of services to fulfill requests. Optimal performance is observed at 32 devices. However, further increasing the number of devices leads to a significant performance decline. This degradation is associated with scheduling overhead, as illustrated in Figure 14. With a larger number of devices, DyOrc's scheduler can become overloaded, causing substantial delays in scheduling requests. Based on these observations, we conducted experiments by dividing our cluster into groups of 32 devices.

9 Related Work

DAG Workflow Scheduler. Workflow orchestration systems, as represented by Airflow [1], Dagster [6], Cadence [4], and Maestro [15], are proposed to schedule DAG workflows. They are typically not designed for machine learning tasks and do not perform optimally for such resource-heavy jobs. For example, they have limited GPU support and cannot exploit hardware like NVLink for efficient data communication between GPUs. Further, task schedulers of these systems do not take the batch processing nature of GPUs into account and may suffer from significant resource underutilization if applied to ML workflows.

Large/Multimodal Model Serving. Various optimization techniques have been proposed for LLM serving, such as continuous batching [64], paged attention [43], grouped-query attention [18] and cross-layer attention [21]. Look-M [60] compresses KV cache to reduce memory usage to accelerate multimodal model inference. Hamed *et al.* [37] introduce early exits to balance prediction performance and efficiency for multimodal document image classification. Lin *et al.* [46] propose to withdraw vision tokens in deep layers to cut computation of vision language models. To ensure general applicability, DyOrc does not incorporate these specific optimizations by default, while these optimizations can be utilized alongside DyOrc.

Serverless ML Serving. Amazon Sagemaker [45] supports serverless DNN serving, which automatically scales serving resources on demand. It is not designed for serving different components in composite machine learning applications. For instance, the maximum request and response payload size for serverless invocations is 4 MB, which is insufficient for tensors between large model components. Gillis [65] automatically partitions a large model into multiple serverless functions. AsyFunc [52] introduces shadow functions that replicate resource-intensive parts of a model to deal with sudden demand rises. ServerlessLLM [34] introduces multi-tier checkpoint loading for faster cold-start. These systems are constraint by the underlying serverless platforms and cannot exploit optimizations such as serialization-free communication.

Serialization-Free IPC. Ray Serve [8] supports serialization-free communication of numpy arrays on the same CPU using shared memory. There are PyTorch proposals [11] for serialization-free IPC for PyTorch tensors on CPU and GPU, based on shared memory and CUDA `cudaIpcGetMemHandle`. DyOrc addresses this issue through its totipotent process design, which executes service functions located on the same physical device within the same address space.

10 Conclusion

We present DyOrc, a system that supports efficient serving of composite machine learning applications with dynamically activated components. DyOrc enables dynamic batching of composite models by dividing them into independently scalable services. With its speculative scheduler, multi-tier message passing, and proactive component loading techniques, DyOrc achieves up to 198% latency improvement over the baselines.

Acknowledgments

This work was supported in part by Alibaba Group through Alibaba Innovative Research (AIR) Program and grants from Hong Kong RGC under the contracts HKU 17205824 and C7004-22G (CRF).

References

[1] 2014. Apache Airflow. https://airflow.apache.org/docs/apache-airflow/stable/project.html. Accessed: 19 Nov. 2023.

[2] 2014. FlatBuffers. https://github.com/google/flatbuffers. Accessed: 8 Oct. 2024.

[3] 2014. XComs. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/xcoms.html. Accessed: 11 Apr. 2025.

[4] 2017. Cadence. https://cadenceworkflow.io/. Accessed: 1 Oct. 2024.

[5] 2017. The Plasma In-Memory Object Store. https://ray-project.github.io/2017/08/08/plasma-in-memory-object-store.html. Accessed: 11 Apr. 2025.

[6] 2018. Dagster. https://dagster.io/. Accessed: 1 Oct. 2024.

[7] 2018. Triton Inference Server. https://github.com/triton-inference-server/server. Accessed: 11 Apr. 2025.

[8] 2019. Ray Serve. https://docs.ray.io/en/latest/serve/index.html. Accessed: 19 Nov. 2023.

[9] 2019. TensorRT. https://github.com/NVIDIA/TensorRT. Accessed: 7 Oct. 2024.

[10] 2021. KServe. https://github.com/kserve/kserve. Accessed: 7 Oct. 2024.

[11] 2021. TorchStore. https://github.com/pytorch/pytorch/issues/64932. Accessed: 8 Oct. 2024.

[12] 2024. ComfyUI. https://github.com/comfyanonymous/ComfyUI/. Accessed: 1 Dec. 2024.

[13] 2024. Coze. https://www.coze.com/. Accessed: 18 Oct. 2024.

[14] 2024. Langflow. https://github.com/langflow-ai/langflow/. Accessed: 1 Dec. 2024.

[15] 2024. Maestro. https://github.com/Netflix/maestro/. Accessed: 1 Oct. 2024.

[16] 2024. tarpc. https://github.com/google/tarpc. Accessed: 7 Dec. 2024.

[17] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*.

[18] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[19] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[20] Hamid Arabnejad and Jorge G Barbosa. 2013. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE transactions on parallel and distributed systems* 25, 3 (2013), 682–694.

[21] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. 2024. Reducing Transformer Key-Value Cache Size with Cross-Layer Attention. *arXiv preprint arXiv:2405.12981* (2024).

[22] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

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

[24] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216.

[25] Yinwei Dai, Rui Pan, Anand Iyer, Kai Li, and Ravi Netravali. 2023. Apparate: Rethinking Early Exits to Tame Latency-Throughput Tensions in ML Serving. *arXiv preprint arXiv:2312.05385* (2023).

[26] Abhishek Das, Satwik Kottur, Khushi Gupta, Avi Singh, Deshraj Yadav, José MF Moura, Devi Parikh, and Dhruv Batra. 2017. Visual dialog. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 326–335.

[27] Jeff Dean. 2021. Introducing pathways: A next-generation ai architecture. *Google Blog* 1 (2021).

[28] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[29] Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first International Conference on Machine Learning*.

[30] E.B. Dynkin. 1965. *Markov processes*. Springer.

[31] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. 2024. ACROBAT: Optimizing Auto-batching of Dynamic Deep Learning at Compile Time. *Proceedings of Machine Learning and Systems* 6 (2024), 14–30.

[32] Shangbin Feng, Weijia Shi, Yuyang Bai, Vidhisha Balachandran, Tianxing He, and Yulia Tsvetkov. 2023. Knowledge Card: Filling LLMs' Knowledge Gaps with Plug-in Specialized Language Models. *arXiv preprint arXiv:2305.09955* (2023).

[33] Martin Fowler. 2012. *Patterns of enterprise application architecture*. Addison-Wesley.

[34] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[35] Yuying Ge, Yixiao Ge, Ziyun Zeng, Xintao Wang, and Ying Shan. 2023. Planting a seed of vision in large language model. *arXiv preprint arXiv:2307.08041* (2023).

[36] Yuying Ge, Sijie Zhao, Ziyun Zeng, Yixiao Ge, Chen Li, Xintao Wang, and Ying Shan. 2023. Making llama see and draw with seed tokenizer. *arXiv preprint arXiv:2310.01218* (2023).

[37] Omar Hamed, Souhail Bakkali, Matthew Blaschko, Sien Moens, and Jordy Van Landeghem. 2024. Multimodal adaptive inference for document image classification with anytime early exiting. In *International Conference on Document Analysis and Recognition*. Springer, 270–286.

[38] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[39] Yuzhou Huang, Liangbin Xie, Xintao Wang, Ziyang Yuan, Xiaodong Cun, Yixiao Ge, Jiantao Zhou, Chao Dong, Rui Huang, Ruimao Zhang, et al. 2024. Smartedit: Exploring complex instruction-based image editing with multimodal large language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8362–8371.

[40] Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. 2024. ESG: Pipeline-Conscious Efficient Scheduling of DNN Workflows on Serverless Platforms with Shareable GPUs. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 42–55.

[41] Sylvain Jeaugey. 2017. Nccl 2.0. In *GPU Technology Conference (GTC)*, Vol. 2.

[42] Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. 2023. Generating images with multimodal language models. *arXiv preprint arXiv:2305.17216* (2023).

[43] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[44] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[45] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. 2020. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 731–737.

[46] Zhihang Lin, Mingbao Lin, Luxi Lin, and Rongrong Ji. 2024. Boosting Multimodal Large Language Models with Visual Tokens Withdrawal for Rapid Inference. *arXiv preprint arXiv:2405.05803* (2024).

[47] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2024. Visual instruction tuning. *Advances in neural information processing systems* 36 (2024).

[48] Costas Mavromatis, Petros Karypis, and George Karypis. 2024. Pack of LLMs: Model Fusion at Test-Time via Perplexity Optimization. *arXiv preprint arXiv:2404.11531* (2024).

[49] Graham Neubig, Yoav Goldberg, and Chris Dyer. 2017. On-the-fly operation batching in dynamic computation graphs. *Advances in Neural Information Processing Systems* 30 (2017).

[50] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[52] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 324–340.

[53] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125* 1, 2 (2022), 3.

[54] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10684–10695.

[55] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical*

conference (USENIX ATC 20). 205–218.

[56] Quan Sun, Yufeng Cui, Xiaosong Zhang, Fan Zhang, Qiying Yu, Yueze Wang, Yongming Rao, Jingjing Liu, Tiejun Huang, and Xinlong Wang. 2024. Generative multimodal models are in-context learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14398–14409.

[57] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2024. Teola: Towards end-to-end optimization of llm-based applications. *arXiv preprint arXiv:2407.00326* (2024).

[58] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[59] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[60] Zhongwei Wan, Ziang Wu, Che Liu, Jinfa Huang, Zhihong Zhu, Peng Jin, Longyue Wang, and Li Yuan. 2024. Look-m: Look-once optimization in kv cache for efficient multimodal long-context inference. *arXiv preprint arXiv:2406.18139* (2024).

[61] Thomas Wolf. 2020. Transformers: State-of-the-Art Natural Language Processing. *arXiv preprint arXiv:1910.03771* (2020).

[62] Xiangyuan Xue, Zeyu Lu, Di Huang, Zidong Wang, Wanli Ouyang, and Lei Bai. 2024. ComfyBench: Benchmarking LLM-based Agents in ComfyUI for Autonomously Designing Collaborative AI Systems. arXiv:2409.01392 [cs.CL] https://arxiv.org/abs/2409.01392

[63] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. 2023. A Survey on Multimodal Large Language Models. *arXiv preprint arXiv:2306.13549* (2023).

[64] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[65] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 138–148.

[66] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[67] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1–14.