QUART: Latency-Aware FaaS System for Pipelining Large Model Inference

Yanying Lin[1,2], Yanbo Li[1,3], Shijie Peng[1,2], Yingfei Tang[1], Shutian Luo[4],
Haiying Shen[5], Chengzhong Xu[6], Kejiang Ye[1,*]
[1] Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
[2] University of Chinese Academy of Sciences [3] Southern University of Science and Technology
[4] Yale University [5] University of Virginia [6] University of Macau

***Abstract*—Pipeline parallelism is a key mechanism to ensure the performance of large model serving systems. These systems need to deal with unpredictable online workloads with low latency and high goodput. However, due to the specific characteristics of large models and resource constraints in pipeline parallelism, existing systems struggle to balance resource allocation across pipeline stages. The primary challenge resides in the differential distribution of requests across various stages of the pipeline.**

We propose QUART, a large model serving system that focuses on optimizing the performance of key stages in pipeline parallelism. QUART dynamically identifies the key stages of the pipeline and introduces an innovative two-level model parameter caching system based on forks to achieve rapid scaling of key stages within seconds. In evaluations with real-world request workloads, QUART reduces average response latency by up to 87.1% and increases goodput by 2.37x compared to the baseline. The experiments demonstrate that QUART effectively reduces tail latency and the average queue length of the pipeline.

***Index Terms*—Pipeline Inference, Large Model, Serverless, Latency Aware**

1. Introduction

Recent advancements in large-scale models [1–4] demonstrate the capability of utilizing a vast number of parameters and complex model structures for general-purpose services. Current reports [5] indicate that a greater number of parameters correspond to superior service capabilities. Consequently, the number of model parameters is continuously increasing. For instance, Google's multimodal large-scale model PaLM2 [4] has reached an impressive 340 billion model parameters. These models are so extensive that a single GPU is insufficient for computation; they need to be partitioned into multiple sub-models and distributed across different devices for computation, utilizing a pipeline parallelism approach [6–9]. Although pipeline inference patterns are widely adopted, achieving more optimization within the pipeline remains a challenging task.

* *Corresponding Author*

Existing objectives of large-scale model serving systems typically aim to provide low-latency, high-throughput inference systems for online requests while achieving high resource utilization. Therefore, current systems often allocate and manage limited system resources from two perspectives: offline optimization [10–14] to squeeze system resources to theoretically optimize performance or online dynamic scaling to provide extreme elasticity [15–20]. Current offline optimized resource scheduling systems often rely on historical request records to predict request loads for resource allocation. Once resources are allocated based on these predictions, the resource configuration usually remains unchanged unless there is a need to serve new models. This approach is suitable for dedicated clusters and may not be easily generalized to shared environments, such as collaborative multi-cloud service providers or dynamic model-serving environments. Hence, there is a growing interest in online resource scheduling to provide real-time resource allocation systems. For instance, recent advancements explore the use of Function as a Service (FaaS) for model inference.

FaaS, or serverless [21–23], is renowned for its high elasticity and pay-as-you-go model, allowing dynamic scaling of its model computation pipeline stages to reduce inference costs. Inference for large models often requires a significant amount of GPU memory, resulting in high costs. Additionally, predicting the arrival of online inference requests can be challenging, leading offline systems to over-allocate computational resources to meet constraints, causing resource wastage [24–26]. The pay-as-you-go strategy of FaaS can significantly reduce inference costs while simultaneously improving the resource utilization of the inference system. However, previous FaaS-based inference systems [27, 28] struggled to respond promptly to large model inference requests within SLO (Service Level Objective). The fundamental reasons behind this issue, apart from the slow loading of large model parameters into GPU [29] compared to millisecond-level response times, involve two key challenges.

Firstly, existing inference systems are tailored for con-

ventional model inference and struggle to cope with the characteristics of large-scale models. Contemporary FaaS systems [18, 27, 28] typically treat the computation of models as a function. However, these systems face challenges in ensuring low latency and high goodput simultaneously when dealing with large-scale models. The fundamental issue is that large models require expensive GPU devices for acceleration, resulting in significant inference overhead. Scaling the model to meet the demands of momentary burst requests is difficult, leading to queuing of inference requests and SLO violations. Secondly, these systems inadequately optimize the pipeline parallel architecture for complex request loads. Recent research advancements have explored the use of Directed Acyclic Graphs (DAGs) [30] to optimize resource allocation in the parallel inference pipeline [17, 31, 32]. However, these methods face challenges in extending to large-scale model inference because they struggle to consider the overall inference performance of the model while optimizing resources for each stage of the pipeline. To address the aforementioned challenges posed by large models in FaaS, we conducted a case study and derived two crucial observations.

First, the request distribution faced by different stages in the pipeline of large-scale model inference may not be consistent. We observe that in the parallelism of large model requests through the pipeline, the call rate to the subsequent stage is perturbed by throughput limitations, leading to an inconsistent request distribution across stages. In the stages of the pipeline, we observe that bursts (measured in terms of Coefficient of Variation, i.e. CV) propagate in an increasing or decreasing manner. This propagation makes it challenging for existing systems to provide optimal service efficiency: methods using a single pipeline stage [10, 11, 14] as the scheduling granularity struggle to cope with short bursts, while methods [27, 28, 33] using the model as the granularity may result in resource inadequacy or wastage in certain pipeline stages.

Second, the performance degradation in large-scale model inference pipelines is often attributed to blocking at certain critical stages. We observe that in the inference pipeline, the accumulation of the waiting queue typically occurs at specific crucial stages. Balancing the performance of critical stages and other stages in a dynamically managed resource system to prevent the generation of pipeline stalls is a new challenge in large-scale model inference.

Based on these observations, we propose QUART, a large model serving system, in which we construct a distributed framework for large-scale model inference that leverages pipeline parallel computing and dynamic scaling in a serverless environment. First, QUART employs latency-aware pipeline critical stage identification, coupled with CV-based propagation inference, to dynamically increase the number of replicas for critical stages, mitigating queuing bottlenecks arising from bursts in request distribution (§5.1). Second, for stages with surplus resources, QUART reclaims excess replicas (§5.2). To prevent additional computational delays caused by recycled resources leading to pipeline stalls, QUART enhances its computational performance through CPU compensation (§5.3).

Furthermore, QUART marks these critical stages and introduces KeysManager (§6.1), a two-stage model parameter caching system designed hierarchically at the pipeline stage level using a fork design. KeysManager achieves sub-second instance creation by caching model parameters in the server's memory. To more efficiently utilize server memory and improve cache hit rates in critical stages, we designed a scheduler (§6.2) based on KL divergence in collaboration with KeysManager. This scheduler aims to distribute critical stages as evenly as possible across the cluster, while also ensuring sufficient GPU memory space is reserved for potential expansion of critical stages.

We implemented QUART based on OpenFaaS and conducted evaluations in real workloads. Our results demonstrate that, compared to the baseline system, QUART effectively reduces average inference latency by 36.7% to 87.1% and simultaneously improves system goodput by 2.19x to 2.37x. To validate the system's performance in more complex workloads, we designed request loads with varying degrees of burstiness. Further details reveal that QUART maintains stable service performance even in highly bursty environments. Compared to the baseline approach, it achieves up to a 32.4x improvement in goodput and simultaneously reduces average response latency by 77.2%. We unveil that QUART effectively mitigates the occurrence of tail latency and significantly reduces the average queue lengths at different stages of the pipeline, resulting in superior performance. The experimental results affirm our insights into critical stage performance.

Contributions. In summary, this paper makes the following contributions:

- We unveil the request propagation across stages in pipeline inference and the hindrance caused by critical stages to pipeline performance.
- We demonstrate that by integrating both model and pipeline-level resource management, a more efficient pipeline can be constructed.
- We showcase that the reduction of pipeline congestion can be achieved through stage-level caching in the pipeline, coupled with cache-aware scheduling.

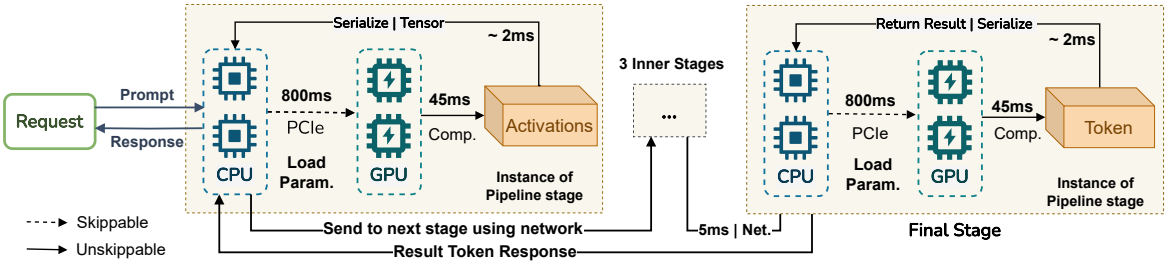


Figure 1: The inference process of a 5-stages pipeline. Stages communicate through the network. Model parameters are cached in both host memory and GPU memory. The model may be temporarily unloaded from GPU memory to host memory. However, when reloaded into the GPU, the KV cache needs to be recalculated. The maximum speed of loading from host memory to GPU memory is theoretically related to the PCIe bandwidth.

2. Background

2.1 Model Inference and Goals

We concentrate on the deployment of an online large model pipeline inference system in cloud and distributed environments. Online systems often contend with stringent SLO requirements, demanding the completion of computations and response delivery within specified deadlines [34]. Given the substantial size of large models, marked by prolonged computation times and the requirement for ample GPU memory to store parameters, the computation of large models is typically executed using pipeline parallelism, also referred to as model parallelism. For instance, the OPT-66B model, which falls into the medium-sized category, requires 132GB of GPU memory to store its parameters. This entails the utilization of four A100 GPUs to handle such a substantial parameter size.

It is common to employ model partitioning to split it into several sub-models, distributing these models across different GPU instances for computation. Fig. 1 illustrates this process: requests, in the form of prompts, pass through a series of pipeline stage instances, ultimately yielding the response token. This token represents the result of the inference and is returned to the source of the request. Network connections link different pipeline stages, and during the transmission process, the intermediate results (activations) of the stages, represented as Tensors, require serialization and deserialization. To achieve this, the pipeline is typically designed with a structure that allows overlap between communication overhead and computational latency.

Moreover, model inference typically requires responses to requests within the range of hundreds of milliseconds to ensure a satisfactory user experience. However, when concurrent requests reach and exceed the system's throughput, it leads to the formation of a waiting queue. The accumulation of this waiting queue significantly increases response latency. Hence, system design commonly involves considering multiple replicas to enhance throughput. For example, a single BERT-21B model exhibits a computational latency close to 200ms on an NVIDIA V100, yielding a theoretical throughput of 5 responses per second (r/s) for a V100 GPU. Due to the idempotence of GPU computations, i.e., the addition of replicas can almost achieve linear throughput growth. For instance, we observe that a stable throughput of 20 r/s can be achieved with 4 replicas of BERT-21B.

2.2 Request Bursts

Online services often face burst requests, which are challenging to predict [10, 11]. Burst requests are characterized by a sudden increase in the number of requests within a short period [22, 35]. During bursts, there could be several times the usual number of requests arriving in a short period. Given the strict SLO, there is a requirement for model instances to scale up to the predetermined level in an extremely short time to meet throughput demands. However, as loading a model from disk to GPU typically takes several seconds, existing methods that scale up based on Queries Per Second (QPS) often lead to SLO violations, as the request peak might subside before scaling is completed. We profiled the loading time of BERT-21B, and despite its 43GB parameter size being loadable into a single NVIDIA A40, the complete loading time from disk is as high as 45 seconds. Scaling up at the time of request arrival evidently cannot meet the SLO for burst requests. Hence, the efficient management of request queues and warmed instances becomes imperative.

3. Case Studies for Pipeline Inference

3.1 Burst Propagation

In a pipeline parallel inference mode, requests traverse each pipeline stage, with the last stage responding to the inference result. During this process, each stage needs to wait for the computation result of the previous stage in order to perform the next step. If the computation time of a certain stage is too long, it will affect the overall execution efficiency of the entire pipeline. In an ideal situation, the computation times of each pipeline stage are similar. That is, inference requests can be evenly distributed to each stage, thereby improving the overall processing speed. However, our observations show that the request distributions received by different pipeline stages

are not consistent within the same period.

We observed that, although the average QPS of each stage is consistent over a longer period, the average queueing time differs significantly due to differences in distribution. We conducted a case study on BERT-21B's 5-stage pipeline using a CV=1 request distribution on our 5-card A40 test platform. We then calculated the CV (coefficient of variation) of each stage's request distribution over 10 minutes. Fig. 2a illustrates that there exist significant differences in CV between different pipeline stages, with a maximum difference of up to 3.5 times. That is, the request distributions received by different stages during the pipeline inference process are not uniform.

Furthermore, we found that the CV of request distributions exhibits an increasing trend across different stages. The main reason is that the throughput of single-stage models is fixed and non-expandable, leading to amplified bursts. This is because the initial CV is relatively small. Different initial CVs and stage numbers would result in more unpredictable and varying propagation patterns. This inspires us to fine-tune the resources at each stage of the pipeline within limited resources to adapt to the characteristics of bursty requests.

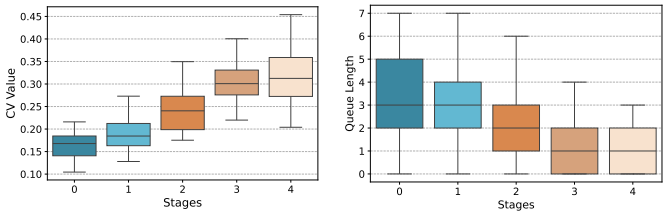
Finding 1

The burstiness of requests in online model inference propagates through its pipeline stages. Moreover, as the request distribution and pipeline structure become increasingly complex, it becomes difficult to predict.

3.2 Pipeline Congestion

The throughput of each stage in the pipeline is usually fixed and is the reciprocal of its computational latency. When the instantaneous request of a model exceeds its throughput, queuing begins to accumulate, and the length of the queue will increase over time. If the pipeline is not scaled up, the growth of the queue will lead to severe performance degradation and increased latency. Due to the difficulty in predicting short-term QPS, queuing is considered a normal phenomenon. In each stage of the pipeline, queuing may exhibit blocking phenomena, affecting the overall throughput of the pipeline. Therefore, scaling up the stages of the pipeline is necessary to ensure that the length of the queue remains within an acceptable range and avoids severe performance degradation. However, finding an appropriate scaling strategy for current model inference systems is difficult. The main challenge comes from balancing congestion avoidance and pipeline performance.

Existing scaling strategies either rely on real-time QPS to dynamically scale up or are based on the QPS of previous stages in the pipeline. The former is unable to timely expand to the predetermined number of replicas due to the slow speed of loading large model parameters into GPU. The latter fails



(a) CV in different stages (b) Queue in different stages

Figure 2: CV propagation in pipeline stages and queue length in different stages. (a) CV in different stages, showing an increasing trend in this case. (b) Queue in different stages, with the key stage having the longest queue.

to resolve congestion in the pipeline, leading to bottlenecks at specific stages (referred to as critical stages). Pipeline congestion will result in severe pipeline stalls, increasing system inference latency.

We designed an experiment to analyze the queuing situation at each stage of the pipeline. We deployed a 5-stage BERT-21B on 5 A40 GPUs without sharing. We observed that the queue length at each stage has a significant difference. Specifically, under an initial request distribution with a coefficient of variation (CV) of 0.1, the average queue length is 3x longer in the shortest queue stage (Fig. 2b). We refer to the longest queue stage as the critical stage, indicating congestion occurs at this stage.

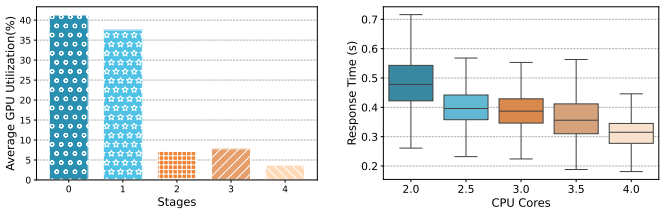
Finding 2

The performance degradation in large-scale model inference pipelines is often attributed to blocking at certain critical stages.

3.3 Stage Resource Underutilization

In deep learning model inference, achieving consistent and uniform throughput across all pipeline stages is crucial for avoiding pipeline stalls. However, this ideal condition is challenging to attain due to fluctuations in online traffic distributions. Current advanced auto-scaling algorithms often scale the entire model uniformly, which means that all pipeline stages are proportionally scaled. While this approach provides stable pipeline performance by ensuring each pipeline unit advances at a balanced computing speed, it struggles to address the issue of inadequate resource utilization arising from changes in request distribution among different pipeline stages.

We analyze a 5-stage BERT-21 pipeline using an initial request distribution with CV=4. We find that the average GPU utilization rates at different stages do not conform (Fig. 3a). This suggests that uniform pipeline scaling will lead to reduced resource utilization due to the burstiness of model requests. The first stage is congested during peak requests, and subsequent stages experience increased burstiness, resulting in queuing. To reduce response latency, parallel computing



(a) GPU Util. in different stages (b) Additional CPUs boost inference

Figure 3: GPU utilization in different stages of pipeline and inference boost with extended CPU cores. (a) GPU utilization in different stages of the pipeline. (b) Inference latency is reduced with additional CPU cores.

requires additional replicas, leading to proportional scaling of the entire pipeline. However, buffering from previous stages can provide a smoother distribution for later pipeline stages, reducing the need for multiple replicas. These replicas occupy GPU memory, making it difficult to allocate these resources effectively to other services. This inspires us to explore non-uniform pipeline stage scaling.

Finding 3

The resource utilization of different pipeline stages is not uniform, leading to underutilization of resources in certain stages.

3.4 Accelerating with Additional CPU Cores

As shown in Fig. 1, cooperative computing between different stages in a pipeline typically involves CPU participation, such as network communication and data loading, tensor serialization, etc. Moreover, modern large models often contain control flows. Current acceleration frameworks, such as CUDA and ROCm, usually switch to CPU execution when encountering control flows during computation. These operations have certain requirements for CPU performance, and their latency is determined by the CPU's performance.

We conducted experiments on an A40 test platform, where all GPUs had GPU direct enabled, and stage activation parameters were transmitted via HTTP/TCP. Fig. 3b illustrates the distribution of inference latency using different numbers of CPUs in the pipeline. Clearly, as the number of CPUs increases, the median inference latency gradually decreases. Compared to a 2-core CPU instance, the 4-core CPU instance reduces the median latency by 33%. This indicates that additional CPU cores can effectively reduce the inference latency in pipelines, particularly in accelerating data transfer to GPUs.

Finding 4

CPU-based data transfer and control flow execution affect inference latency. Additional CPU cores can accelerate these operations, reducing inference latency.

4. QUART Design

4.1 Overcoming Requests Fluctuation

We capitalize on three key observations to overcome request distribution fluctuations.

Pipeline congestion and resource correctness. We observe that the propagation of requests within pipeline units is not uniformly conducted. Throughout the process of completing an inference request, computations traverse different pipeline units sequentially, following a linear or DAGs (Directed Acyclic Graphs) structure, until reaching the final pipeline stage. In an ideal scenario where requests uniformly arrive, filling the entire pipeline, the pipeline can operate at maximum throughput. In this situation, the queue consistently maintains a low water level: as requests arrive, they are promptly executed, and upon completing the current request, a new one immediately follows. Consequently, the average waiting time is reduced to zero.

However, requests in online services typically exhibit burstiness. Bursty requests inevitably result in some degree of queuing. Moreover, due to the varied execution times and pipeline stalls influenced by different models, the CV differs across the units of an inference pipeline for a given model. The buildup of queue length often leads to extended service response times, potentially causing violations of SLO. In response to this challenge, our observation indicates that the inference pipeline of the model tends to accumulate predominantly in earlier stages. This motivates the development of a pipeline congestion detection mechanism and an online scaling mechanism that, through resource allocating correction, aims to reduce response latency. This method offers an approach to dynamically increase resources for congested stages in the pipeline through delayed feedback. By augmenting the number of replicas, the queues in the pipeline stages are dispersed, resulting in reduced latency. This method achieves a cost-effective reduction in latency compared to increasing the number of entire pipeline replicas by allocating resources to a few stages.

CV propagation and Replica Smoothing. Online requests for models are sequentially executed in a directed manner through the stages of the pipeline. Each stage of the pipeline must wait for the completion of its predecessor stage, taking its output as an activation parameter for the model's input. Over an extended period, the average QPS for each pipeline stage remains relatively consistent unless there is a failure in the middle. Therefore, it is common to allocate equal resources to different pipeline stages. However, we observe that, in reality, the resource utilization of certain pipeline units is significantly lower than other stages.

To address this, we observe that the burstiness of requests in online model inference propagates across its pipeline stages.

The CV is commonly used to quantify the burstiness of request distributions. As the throughput of each pipeline stage is limited, it constrains the burstiness of subsequent pipeline stages within a specific range. This gives rise to a phenomenon where the CV propagates across the pipeline stages, exhibiting variation between different stages and presenting either an increasing or decreasing trend. We applied varying CVs to an 5-stage pipeline of BERT-21B (split into 8 sub-models) and observed the request distribution characteristics across its different pipeline stages. We notice that, with lower CV values, the CV of request distribution increases as we move towards later stages in the pipeline. For instance, when the model has a CV of 0.1, the CV for pipeline stages gradually increases from 0.03 to 0.25, demonstrating a noticeable trend of incremental rise as we move towards later pipeline stages. This drives us to refine the management of resources for different stages of the model pipeline by constructing a model for burstiness propagation in the pipeline. The underlying concept is that pipeline stages with a higher CV require more replicas to provide high throughput, while stages with a lower CV can allocate a portion of their resources to stages with a higher CV to reduce the overall latency of the pipeline.

CPU accelerating and additional resources. Each stage of the model pipeline represents a sub-model scheduled for computation on different server stages. While the primary computation involves tensor calculations on GPUs, a considerable amount of support from other devices, such as network cards, memory, CPUs, etc., is still essential throughout a stage's complete cycle (Fig. 1). Among these, the CPU plays a crucial role. Firstly, different stages of the pipeline communicate through network communication, involving the serialization and deserialization of activation propagation, as well as the overhead of the operating system network stack during network transmission. These are all aspects that heavily involve the CPU. Secondly, there is the collaboration and context switching between the CPU and GPU during model inference. Modern deep learning models often include control flow to enhance performance. These control flows involve logical computations that typically trigger context switches to CPU computation in existing computing frameworks, making CPU performance crucial.

When the number of replicas for a particular stage in the pipeline is reduced, requests may become concentrated and distributed among the remaining replicas. In other words, the communication overhead for these replicas may increase, potentially leading to a situation where communication and computation time cannot overlap, possibly resulting in pipeline stalls. To address this, we conducted a case study analyzing the performance improvement through CPU parallelism. In our experiments on a 5-stage pipeline of BERT-21B, we observed

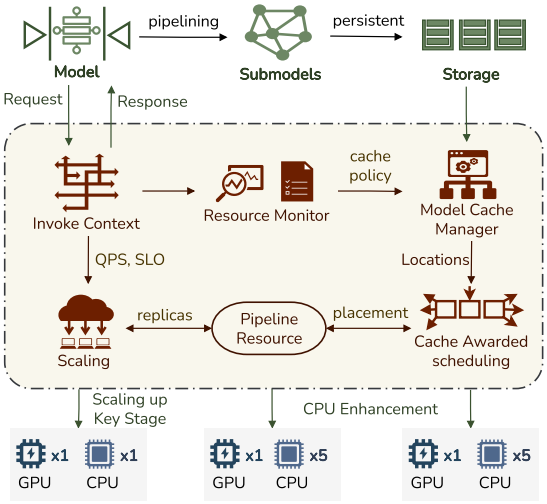


Figure 4: The architecture of QUART. It consists of four key components: replica corrector, pipeline stage smoother, CPU compensator, and cache-aware scheduler.

that, with the same GPU scale, instances with a higher number of CPU cores significantly outperformed those with fewer cores. This inspired us to leverage CPU parallelism to alleviate the additional overhead caused by communication costs and control flow context switching.

4.2 Mitigating Replicas Ready Delay

When model requests experience a sudden increase, additional replicas are required to enhance throughput. However, due to the substantial size of large models, even though they are split into multiple smaller sub-models, the time it takes to load them from disk makes it challenging to meet SLOs in response to burst requests. We observed that loading the model from the server's main memory to the GPU memory incurs a relatively small overhead. Fig. 1 illustrates that loading a sub-model of an 8-stage BERT-21B from CPU to GPU takes only 800ms. This latency is significant, inspiring us to explore methods of caching models in the server's memory. However, caching all models in memory simultaneously is impractical. The most critical factor is that the CPU memory is not abundant enough to cache all models on every server.

To address this, we observed that rapidly scaling up a particular pipeline stage can effectively alleviate the performance challenges caused by burst traffic. Firstly, in §4.1, we analyzed that burst traffic typically congests at a specific stage, and bursts in pipeline stages usually exhibit a gradient propagation effect. Furthermore, we summarized the mechanisms of burst propagation within the pipeline. Therefore, we conclude that the key to promptly scaling up to meet SLOs during burst requests lies in whether several key pipeline stages can be expanded to the target scale promptly.

Model computations require a copy of parameters stored in memory. Having multiple replicas of the same model on a

single GPU is entirely redundant. However, a server typically houses multiple GPU accelerators, which share the server's memory. Therefore, the key principle of our design is that the server's memory can serve as a cache for the model, allowing for rapid forking of a replica to a new GPU. This approach eliminates the lengthy wait time associated with loading the model from disk.

4.3 System Architecture

QUART strategically leverages key observations in burst propagation and delay feedback in its design. It incorporates an online pipeline stage replica manager and a cache-aware instance scheduler. The architecture of QUART (Fig. 4) consists of four key components: a replica corrector (§5.1), pipeline stage smoother (§5.2), CPU compensator (§5.3), and a cache-aware scheduler (§6). The replica corrector dynamically adjusts the number of replicas for each stage of the pipeline using a PID control method based on the pipeline queue status and feedback on response latency. Subsequently, the pipeline stage smoother computes the resources of pipeline units based on the propagation of request distribution and reclaims a certain number of replicas to improve resource utilization while ensuring throughput. For the reclaimed replicas, the remaining instances in the pipeline stages may experience increased communication pressure due to rising requests. The CPU compensator, employing delay feedback, adds a specific number of CPUs to enhance parallelism.

5. Resource Correct and Smooth

5.1 Replicas Corrector for Scaling Up

The corrector determines the pipeline stages that need scaling up and the target number of replicas through two steps. Firstly, it identifies potential bottleneck stages based on queuing theory and latency considerations. The pipeline comprises n processing units, each having a specific number of replicas r_i, service rate μ_i, and arrival rate λ_i. The replica count indicates the number of requests that can be processed in parallel by that unit, the service rate represents the rate at which a single replica completes requests (throughput), and the arrival rate λ_i signifies the rate at which requests arrive at that unit. Considering the potential existence of multiple replicas at each stage of the pipeline, we utilize the M/M/c queuing model to estimate the queuing delay T_i for each processing unit. The M/M/c queuing model is a classical queuing model that assumes a Poisson arrival process and an exponential service time distribution. The queuing delay T_i for each processing unit is calculated using Eq. 1.

$$T_i = \frac{P_0 \lambda_i}{r_i \mu_i (\mu_i - \lambda_i / r_i)} + \frac{1}{\mu_i} \tag{1}$$

where P_0 represents the probability that the stage is idle when there are no requests. It is calculated by solving the balance equations.

Subsequently, we determine the target queuing delay T_{target} based on the SLO setting for each stage of the pipeline. If the actual queuing delay T_i exceeds the target value T_{target}, increase the number of replicas r_i for unit i:

$$r_i'(t) = r_i(t) + \Delta r \tag{2}$$

The dynamic correction of the replica count is achieved through the utilization of a PID controller. This controller, named for its three fundamental control components—Proportional (K_p), Integral (K_i), and Derivative (K_d), is an extensively employed control system element. The integration of these three control modes is employed to fine-tune the system's output, ensuring more efficient attainment of the predetermined objective or setpoint, as Eq. 3 illustrates.

$$\Delta r = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt} \tag{3}$$

This approach allows for the optimization of the performance of pipeline processing units, particularly in systems with significant demand fluctuations. It effectively balances the load and response time.

5.2 Pipeline Stages Smoothing

The number of model replicas is calculated based on QPS, and after adjusting the replica count for congested pipeline stages using a Corrector, some replicas exhibit low resource utilization. The proliferation of replicas aims to mitigate pipeline stalls caused by sudden surges in traffic. However, the propagated burst of traffic within the pipeline may result in diminished spread. In other words, certain replicas in pipeline units become redundant. To enhance resource utilization, optimizing GPU allocation for critical paths, we employ a mechanism based on CV propagation to identify surplus resources in pipeline stages and use the Smooth method to gracefully reclaim resources.

Initially, we employ the Attention mechanism to construct patterns of CV propagation in the pipeline. For each stage i in the graph $G = (V, E)$, its coefficient of variation CV_i can be computed based on the feature distribution of neighboring stages of i. The coefficient of variation is the ratio of the standard deviation to the mean and is commonly used to measure relative dispersion. For stage i, CV_i can be defined as $CV_i = \frac{\sigma(\mathcal{N}_i)}{\mu(\mathcal{N}_i)}$, where $\sigma(\mathcal{N}_i)$ and $\mu(\mathcal{N}_i)$ represent the standard deviation and mean, respectively, of the request distribution of neighboring stages for stage i.

We incorporate CV into the Attention mechanism. When computing the updated features for stage i, we can integrate CV_i into the calculation of attention coefficients. This implies that the attention coefficient for stage i is influenced not only by its

similarity to its neighbors but also by the variability of features in its neighboring stages. For stage i and its neighboring stage j, the attention coefficient α_{ij} can be adjusted as follows:

$$\alpha_{ij} = \frac{\exp(\text{LReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{x}_i \,||\, \mathbf{W}\mathbf{x}_j]) + \theta \cdot CV_i)}{\sum_{k\in\mathcal{N}(i)} \exp(\text{LReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{x}_i \,||\, \mathbf{W}\mathbf{x}_k]) + \theta \cdot CV_i)} \tag{4}$$

where $\mathbf{W}$ is a weight matrix, $\mathbf{x}_i$ and $\mathbf{x}_j$ are the feature vectors of stages i and j, respectively, $\mathbf{a}$ is a weight vector, and θ is a hyperparameter that controls the influence of CV_i on the attention coefficient. Hence, the updated feature $\mathbf{x}'_i$ for stage i is formed by the weighted sum of neighboring features, with weights provided by the attention coefficients α_{ij}:

$$\mathbf{x}'_i = \sum_{j\in\mathcal{N}(i)} \alpha_{ij}\mathbf{W}\mathbf{x}_j \tag{5}$$

During the training of this Attention model, we include θ as a parameter to be learned. Consequently, we obtain a CV prediction model $\mathbf{f}(\cdot)$ tailored for different models and pipeline stages. Thus, for stage i, the predicted CV value $\hat{CV}_i$ can be expressed as:

$$\hat{CV}_i = \mathbf{f}(\mathbf{x}'_i) \tag{6}$$

To determine how much resources can be reclaimed for a pipeline stage, we employ the CV of this stage along with the resource utilization of its upstream and downstream stages to smoothly compute the new allocation of resources. Specifically, we utilize Laplace smoothing, with CV serving as the primary basis for its smoothing coefficient β_i. The actual resources y'_i required for stage i are given by:

$$y'_i = \beta(CV_i)\hat{y}_i + (1-\beta(CV_i)) \sum_{j\in\mathcal{N}(i)} \frac{\hat{y}_j}{|\mathcal{N}(i)|} \tag{7}$$

where $\hat{y}_i$ represents the total allocated resources after expansion, and $\sum_{j\in\mathcal{N}(i)} \frac{\hat{y}_j}{|\mathcal{N}(i)|}$ denotes the average resource demand of all neighbors of stage i. $\beta(CV_i)$ is a buffering function that dynamically adjusts parameters based on the coefficient of variation CV_i of stage i. It determines the relative importance of its demand and the demands of its neighbors in the resource allocation process.

5.3 CPU Compensate

Following the reduction in model replicas, the gateway redistributes traffic among the remaining instances of pipeline stages. This results in an augmented load on instances related to CPU pressures (§4.1). While the Smoother calculation guarantees no SLO violations under predictable request distributions, optimizing its communication and CPU parallel capabilities is still imperative for reducing response latency and avoiding potential pipeline stalls.

We initially conducted offline profiling of model computation time, communication volume, and communication time. Subsequently, we measured the model's response time on instances with varying CPU counts. Previous research progress has indicated that GPU computation time can be easily estimated based on the parameter quantity. Therefore, predictions can be made based on the model's parameter quantity and request queue length. Concurrently, leveraging CPU parallel profiling allows us to anticipate queueing time and latency modifications following changes in CPU count. Building upon this, we constructed a linear regression model to predict the number of CPU cores required for instances.

Algorithm 1: Correct and Smooth in Pipeline

Input: λ_i: The arrival rate for stage i.
μ_i: The service rate of each replica in stage i.
r_i: The current number of replicas for stage i.
T_{target}: The target queue delay for the system.
R_{total}: The total resource constraint for the system.
K_p, K_i, K_d: Parameters of the PID controller, representing the proportional, integral, and derivative gains, respectively.
$r_{i,\text{min}}$: The minimum required number of replicas for stage i to function properly.
$CV_{\text{pre},i}$: CV of the requests for the precursor units of i.
Output: Updated replicas r'_i for each stage i

Function `Correct` (λ_i, μ_i, r_i, T_{target}) **:**
  $e(t) \leftarrow T_{\text{target}} - T_i(t)$
  $\Delta r \leftarrow K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$
  $r'_i \leftarrow r_i + \Delta r$
  return r'_i
Function `Smooth` (r_i, γ, $r_{i,min}$, $CV_{pre,i}$) **:**
  $CV_i \leftarrow$ CV propagation from precursor stage $CV_{\text{pre},i}$;
  $r'_i \leftarrow$ calculate replicas using CV_i and QPS μ_i
  if $r'_i < r_{i,min}$ **then**
    $r'_i \leftarrow r_{i,\text{min}}$;
  if $r'_i < r_i$ **then**
    AllocateCPU(λ_i, μ_i, r_i, C_i, T_{target})
  return r'_i;
Function `AllocateCPU` (λ_i, μ_i, r_i, C_i, T_{target}) **:**
  for *each stage* i **do**
    Compute T_i based on λ_i, μ_i, r_i, C_i
    if $T_i > T_{target}$ **then**
      $C'_i \leftarrow$ Increase C_i based on regression
    Ensure $C'_i \leq CPU_{\text{max}}$
    if $C'_i > CPU_{max}$ **then**
      SLO may be violated, increase r_i
      Correct(λ_i, μ_i, r_i, T_{target})
      return
    return C'_i for each stage i

6. Online Cache-Aware Scheduling

The initialization of models typically demands a substantial loading time. However, the swift transfer of parameters from memory to GPU, often completed within seconds, is a notable efficiency. Nevertheless, memory is typically limited, and replicating multiple identical copies on the same GPU provides negligible performance improvement. In §4.2, we conducted an analysis, revealing the challenges associated with this scenario. Consequently, the design of an effective caching system in coordination with model scheduling becomes crucial. Based

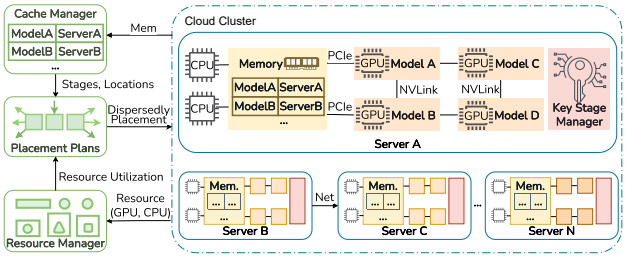


Figure 5: Cache-Aware Scheduling. Key stages of a model would be prioritized to be scheduled on servers with available cache parameters.

on our observations, we collaboratively designed a scheduling system with cache awareness, coordinating with the Corrector. This system is intended to swiftly scale the key stages of the pipeline by leveraging a small amount of model caching at the memory level.

6.1 Cache Manager

We collaborated with Corrector to design a list of key stages in the pipeline. When Corrector identifies a key stage and adds replicas to it, it is appended to the list. As illustrated in Fig. 5, the cache manager keeps track of the servers where these key stages are located. All GPU accelerators on the same server share their memory, and a common replica of the model parameters computed in the GPU resides in memory. To fully leverage the parameters cached in memory, we maintain a scheduling history of key stages in the cache manager, indicating which machines these stages have been scheduled on. Simultaneously, we track the validity of these cached replicas. To prevent cache inconsistency, each machine's memory contains a local copy of its cache, and we have implemented a cache invalidation hook.

Initially, a Key Stages Manager (KeysManager), responsible for managing key stages deployed on a specific server, is deployed at the server level. In the KeysManager, upon the scheduling of a stage to the server, we utilize the Copy-on-Write (COW) mechanism to *fork* a subprocess of the key stage, recording the subprocess's PID. The critical design aspect of forking lies in retaining data in the server's memory after the stage is evicted. A crucial challenge arises as forking also duplicates GPU memory unnecessarily. To address this, we control whether parameters are synchronized to the GPU by modifying environment variables before forking, skipping operations such as *torch.cuda.to_device*. Simultaneously, the KeysManager monitors the machine's memory capacity. When the memory usage surpasses a predefined threshold (set to 85% in this work), the KeysManager employs the LRU algorithm to terminate cached processes, reclaiming memory resources.

When a key stage is scheduled to this server, if the KeysManager identifies the presence of a PID for a stage with identical parameters, it only needs to modify environment variables. Subsequently, using traditional forking (non-COW mode), a process can be duplicated for the normal inference process. In this mode, a duplicate set of parameters is created, and simultaneously, this copied set of parameters is synchronized to the GPU. Although this incurs a certain overhead, it ensures the security and privacy of computations.

6.2 Dispersedly Placement

Previous studies [10, 36, 37] have investigated the potential performance interference and degradation caused by GPU multiplexing. Simultaneously, excessive concentration of key stages can diminish the potential for scaling these stages: instances of the stage might exhaust GPU memory. To address this, we devised a sparse scheduling algorithm specifically tailored for key stages.

In situations where resource competition is not intense, the scheduling strategy can be optimally crafted. key stages are given precedence on servers with the most available GPUs, while other stages are densely arranged. This ensures that servers hosting key stages maintain a notable number of idle GPUs. In instances requiring rapid scaling of key stages, copies can be efficiently transferred from server memory to new GPUs, with less time spent on loading from disk.

In situations of intense resource contention, implying that key stages cannot fully occupy an entire machine, we employ KL divergence for quantifying the placement of critical models. The KL divergence is a measure of the difference between two probability distributions. In our case, it is used to measure the difference between the distribution of key stages and the distribution of all stages. Assuming we have N GPU instances (stages) to be placed on a grid consisting of M possible positions (servers), the initial state P is a matrix of dimensions $N \times M$, where P_{ij} represents the probability of the i-th instance being placed at the j-th position. The target state Q is also a matrix of dimensions $N \times M$, representing the ideal placement strategy. Under the sparse placement strategy, we aim for instances to be as dispersed as possible. Therefore, Q can be designed as a uniform distribution, meaning the probability for each instance at each position is equal. The KL divergence is defined as:

$$D_{KL}(P_i||Q_i) = \sum_{j=1}^{M} P_{ij} \log \frac{P_{ij}}{Q_{ij}} \tag{8}$$

Upon obtaining information about the number of key stages, we can directly compute the optimal Q. To minimize KL divergence, we can utilize gradient descent or other optimization algorithms. In each iteration, we update P using the gradient of KL divergence. The gradient for P_{ij} is:

$$\frac{\partial D_{KL}}{\partial P_{ij}} = 1 + \log \frac{P_{ij}}{Q_{ij}} \tag{9}$$

TABLE 1: Model details. The execution time is the time taken for the model to perform standalone inference on a GPU.

Model	Size(GB)	Opers.	Token	Exec.(ms)
Whisper-9B [38]	2.9	677	x	74
BERT-21B [3]	43	1195	128	137
LLAMA-7B [1]	14	225	1024	243
OPT-66B [2]	132	581	128	278
WideResnet[39]	16	295	x	646

The gradient descent algorithm is as follows:

$$P_{ij} \leftarrow P_{ij} - \alpha \frac{\partial D_{KL}}{\partial P_{ij}} \tag{10}$$

where α is the learning rate. The algorithm iterates until the KL divergence converges. Upon computing the new P, we can iteratively schedule key stages until the KL divergence fails to meet its threshold or new key stages appear. Subsequently, the next iteration of P calculation commences.

7. Implementation

Built on the foundation of OpenFaaS, we implemented QUART by installing its components as plugins into Kubernetes. Interaction with OpenFaaS is facilitated through its open-source tool, faas-cli. The entire implementation comprises 5,000 lines of code, with 2,000 lines dedicated to the realization of KeysManager and the scheduler, and the remaining 3,000 lines dedicated to the implementation of online resource management, including Correct and Smooth. Additionally, we implemented tools for latency collection, system performance monitoring, and an HTTP request load generator with its benchmark using around 800 lines of code.

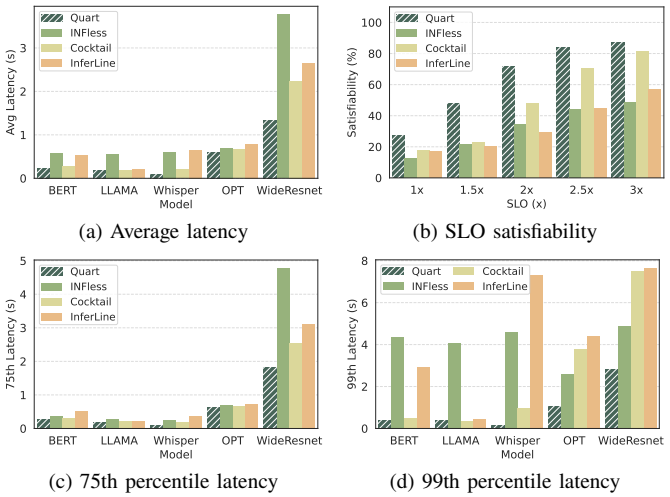
8. Evaluation

8.1 Experimental Setup

We conducted experiments in a cluster comprising a total of 24 servers. All servers were equipped with Ubuntu 22.04 and essential infrastructure such as *containerd*, *CNI*, GPU drivers, etc. The experimental cluster was built on Kubernetes v1.24 and above, encompassing 12 CPU servers and 12 GPU servers. For the CPU servers, a configuration of 40 cores and 64GB of memory was employed to deploy middleware supporting inference. The GPU servers, on the other hand, featured 108 cores, 1024GB of memory, and 4 GPUs.

Large Models. We selected five widely used large models, including the large language model LLAMA [40], BERT [3], GPT [2], and the DNN models Whisper [38] and Wide-Resnet [39], detail shown in Table 1. To prevent pipeline stalls caused by uneven model segmentation, we evenly divided these models into eight pipeline stages based on DeepSpeed. Each pipeline stage is deployed using a Docker instance, serving as a function in OpenFaas.

Baseline. This work primarily focuses on comparing INFless [27], InferLine [17] and Cocktail [16]. INFless aims to leverage serverless architecture for machine learning model inference. It incorporates a series of latency and throughput optimization algorithms, synergizing components like AutoScaler and Cold Start Manager, making it closely related to the design of this work. InferLine adopts a DAG design to infer deep learning models in a pipeline fashion. InferLine's approach to resource allocation and dynamic scaling based on latency awareness is comparable to the content in §5 of this paper. Cocktail adopts an adaptive resource management framework, and we integrate it with the pipeline to assess the resource management performance of QUART.



(a) Average latency (b) SLO satisfiability

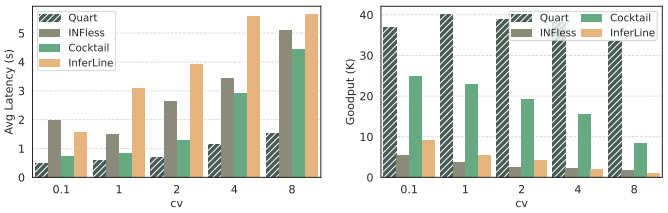
(c) 75th percentile latency (d) 99th percentile latency

Figure 6: Latency and SLO satisfiability for different models. (a) Average latency. (b) SLO satisfiability with different SLO sets to a multiple of each model inference time. (c) 75th percentile latency. (d) 99th percentile latency.

Workload. We employed two types of workloads for our experiments. One is a real-world workload used for production environment performance validation, sourced from Microsoft Azure Function's publicly released production-level trace (MAF). MAF encompasses interleaved processing of various workloads, including bursty and fluctuating workloads. The other type involves fluctuating loads generated based on CV, utilized to assess the system's performance boundaries under extreme conditions.

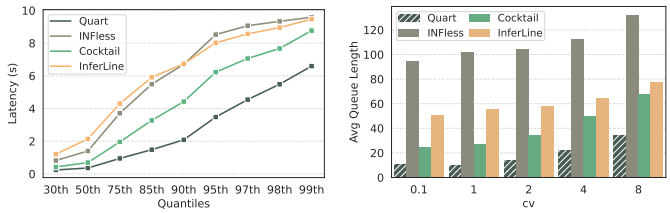
8.2 Overall End-to-End Performance

We compared the performance of QUART and the baseline method under real workloads. Fig. 6 illustrates the comparison of response latency and SLO satisfaction rates. Across various models, QUART exhibits an average reduction in response latency of 87.1%, 70.5%, and 36.7% compared to INFless, InferLine, and Cocktail, respectively (Fig. 6a). Specifically, compared to the optimal method in the baseline, Cocktail, QUART achieves an average reduction in response time of 16.2%, 9.1%, 10.2%, 61.2%, and 41% in BERT, LLAMA, Whisper, OPT, and WideResnet, respectively.



(a) Average latency (b) Goodput

Figure 7: Average latency and goodput for different models in different request distribution with cv=0.1,1,2,4,8. (a) Average latency. (b) goodput.



(a) Latency with diff. quantiles (b) Average queue length

Figure 8: Latency and queue length for different models in different request distribution. (a) Latency with different quantiles. (b) Average queue length.

For SLO satisfaction, we use the model's profiled inference time (1x) as the baseline to quantify the online performance of different methods. We observe the completion rates of QUART and the baseline methods for inference times ranging from 1x to 3x as SLO (Fig. 6b). Goodput, representing the count of effective token responses, is used to quantify SLO. At 1.5x, QUART achieves a SLO satisfaction rate of 47.5%, showing improvements of 2.19x, 2.37x, and 2.09x compared to INFless, InferLine, and Cocktail, respectively.

8.3 Performance in More Complex Workloads

To showcase the performance of QUART in a more complex request environment, we generated request sequences and workloads based on CV and referenced Inter-Arrival Time (IAT). We generated five types of request distributions with CV values of 0.1, 1, 2, 4, and 8. These distributions represent varying levels of burstiness in requests, ranging from low to high. A higher CV indicates stronger burstiness, making it more prone to triggering pipeline congestion at key stages. In this experiment, to measure the extreme performance of the compared methods, we configured a large QPS and set the initial replicas for all scaling to 1. Additionally, we imposed a lenient SLO constraint. This configuration indicates that longer queueing times are permissible at each pipeline stage, leading to increased queue lengths and amplification of congestion. Fig. 7 illustrates the comparison of average response latency and goodput across different request distributions.

QUART maintains a relatively stable average response time across different distributions (Fig. 7a). Among the baseline methods, Cocktail exhibits performance close to QUART for small CV values (CV=0.1, 1). However, as CV increases, Cocktail shows a noticeable upward trend in average latency, reaching 2.93x at CV=8. Overall, compared to INFless, InferLine, and Cocktail, QUART achieves an average reduction in average response latency of 69.8%, 77.2%, and 47.8% across all distributions. Across different request distributions, QUART exhibits average improvements of 13.5x, 14.4x, and 2.3x in goodput compared to INFless, InferLine, and Cocktail, respectively (Fig. 7b). Particularly, in high request burstiness loads with CV=8, QUART achieves notable improvements of 17.9x, 32.4x and 3.9x goodput.

8.4 Benefit from Key Stages Manager

QUART mitigates pipeline congestion by rapidly scaling the key stage, thus reducing pipeline bottlenecks. We observe the benefits of the key stage through two critical metrics: different quantiles of latency and the average queue length at each stage (Fig. 8). Tail latency serves as an effective metric for observing pipeline congestion. We compare the different response latency quantiles (Fig. 8a) of QUART and the baseline to observe the comprehensive outcomes of QUART in alleviating cold start at pipeline stages and avoiding congestion at key stages. QUART consistently performs optimally in all presented latency quantiles. INFless, lacking the use of pipeline parallelism, faces prolonged model scale-up and scale-down times, leading to an inability to meet SLO. InferLine and Cocktail, while employing pipeline-like mechanisms, are constrained by congestion at key pipeline stages, limiting their performance potential. Further insights can be gleaned from the analysis of model queue average lengths in Fig. 8b. QUART maintains relatively short queue lengths across all CV values. INFless, not utilizing a pipeline form, experiences longer inference times for individual models, leading to extended queue buildup. For instance, INFless achieves an average queue length that is as high as 10.1x compared to QUART. Cocktail and InferLine, lacking optimization for overall pipeline performance, miss out on achieving the optimal queue lengths.

9. Related Work

Model Serving Systems. In recent years, there has been significant attention on GPU cluster performance optimization for large-scale model serving, with numerous systems proposed to address various aspects of large model serving. Early general-purpose model inference systems [15, 17, 19, 27, 33] have focused on batch processing, caching, elasticity, and scheduling to meet the service demands of single or multiple models. In the latest advancements in large-scale model serving systems, AlpaServe [10] adopts model parallelism for statistical multiplexing, designing an optimized serving architecture and scheduling strategy based on historical request distributions. Shepherd [11], and SpotServe [14] introduce preemptive serving. vLLM [41], combines KV cache offloading to enhance

system throughput.

Severless for Deep Learning. Recently, many studies have considered the cost aspect of large model computing systems [27–29, 42]. Serverless architecture has been shown to help reduce costs and improve system resource utilization through dynamic scaling. For instance, ElasticFlow [42] designed a serverless-like mechanism to provide a framework for model training. INFless [27] proposed a feasible solution for inference in serverless environments, while Tetris [28] optimized model parameter redundancy across multiple serverless instances. The recent ServerlessLLM introduced parallel loading and local scheduling mechanisms to enhance system elasticity. Quart, on the other hand, provided fine-grained resource management strategies at the pipeline stage. In latency-aware resource management systems, previous work, such as MArk [18], introduced SLO-aware system resource scheduling and scaling strategies [43, 44] to ensure performance meeting SLO constraints. QUART focuses on the inference pipeline, leveraging a detailed analysis of the performance of pipeline stages. Through the integration of cache and scheduling methods, it aims to reduce pipeline congestion and enhance performance.

10. Limitation and Future Work

Despite the enhancement of overall system performance achieved by addressing critical stages, large-scale model pipeline parallel inference applications in shared environments remain challenging. The presence of pipeline stalls, caused by factors such as GPU sharing and device heterogeneity, continues to impact pipeline performance, leading to potential performance degradation. Additionally, there is considerable exploration space for mechanisms that leverage more abundant memory caching of parameters by models in large-scale systems. In this paper, due to device limitations, we only explored the use of local caching to accelerate model loading, leaving substantial potential untapped, particularly in leveraging the high-speed network, RDMA and distributed caching mechanisms. Both of these directions will be our primary focus for further exploration.

11. Conclusion

In large-scale model systems employing pipeline parallelism for inference, the disparate distribution of requests across stages leads to concurrent reductions in resource utilization and performance. We introduce QUART, a novel pipelining model serving system. Leveraging latency-aware strategies, QUART effectively mitigates the challenges posed by key stage congestion induced by bursty requests, leading to queue accumulation. It achieves this through a combination of techniques, including leveraging key stage parameter caching, resource scheduling based on key stage positions, resource reclamation from nodes with surplus resources, and compensatory measures utilizing CPU resources. Evaluation with real-world workloads demonstrates that QUART successfully alleviates tail latency and reduces the system's average response latency by up to 87.1%.

Acknowledgment

This work is supported by the National Key R&D Program of China (No. 2021YFB3300200), National Natural Science Foundation of China (No. 62072451, 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), Shenzhen Basic Research Program (No. JCYJ20220818101610023) and Zhejiang Lab Open Research Project (NO.K2022NB0AB01).

References

[1] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozi‘ere, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," Feb. 2023.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," Jul. 2020.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," May 2019.

[4] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen *et al.*, "Palm 2 technical report," Sep. 2023.

[5] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," Jan. 2020.

[6] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, Georgia, Nov. 2020, pp. 1–16.

[7] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: Scalable, low-cost training of massive deep learning models," in *Proceedings of the Seventeenth European Conference on Computer Systems*, New York, NY, USA, Mar. 2022, pp. 472–487.

[8] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, New York, NY, USA, Mar. 2019, pp. 1–17.

[9] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for dnn parallelization," in *Advances in Neural Information Processing Systems*, 2021, pp. 24 829–24 840.

[10] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez *et al.*, "Alpaserve: Statistical multiplexing with model parallelism for deep learning serving," in *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12,*

2023, 2023, pp. 663–679.

[11] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "Shepherd: Serving dnns in the wild," in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, 2023, pp. 787–808.

[12] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, Feb. 2021, pp. 431–445.

[13] C. Li, Z. Yao, X. Wu, M. Zhang, C. Holmes, C. Li, and Y. He, "Deepspeed data efficiency: Improving deep learning model quality and training efficiency via efficient data sampling and routing," Feb. 2023.

[14] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," Nov. 2023.

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[16] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Cocktail: A multidimensional optimization for model serving in cloud," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1041–1057.

[17] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "Inferline: Latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, New York, NY, USA, Oct. 2020, pp. 477–491.

[18] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USA, Jul. 2019, pp. 1049–1062.

[19] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for transformer-based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[20] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, 2021, pp. 397–411.

[21] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, pp. 1522–1539, Mar. 2023.

[22] M. Shahrad, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.

[23] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, New York, NY, USA, Oct. 2020, pp. 30–44.

[24] W. Zhang, B. Chen, Z. Han, Q. Chen, P. Cheng, F. Yang, R. Shu, Y. Yang, and M. Guo, "Pilotfish: Harvesting free cycles of cloud gaming with deep learning training," in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, 2022, pp. 217–232.

[25] A. Fuerst, S. Novakovi'c, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini, "Memory-harvesting vms in cloud platforms," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne Switzerland, Feb. 2022, pp. 583–594.

[26] S. Choi, T. Kim, J. Jeong, R. Ausavarungnirun, M. Jeon, Y. Kwon, and J. Ahn, "Memory harvesting in multi-gpu systems with hierarchical unified virtual memory," in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, 2022, pp. 625–638.

[27] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: A native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne Switzerland, Feb. 2022, pp. 768–781.

[28] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, 2022.

[29] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "Serverlessllm: Locality-enhanced serverless inference for large language models," Jan. 2024.

[30] S. Shen, L. Hou, Y. Zhou, N. Du, S. Longpre, J. Wei, H. W. Chung, B. Zoph, W. Fedus, X. Chen *et al.*, "Mixture-of-experts meets instruction tuning:a winning combination for large language models," Jul. 2023.

[31] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, 2022, pp. 303–320.

[32] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, Virtual Event USA, Oct. 2020, pp. 1–15.

[33] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving dnns like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, 2020, pp. 443–462.

[34] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with sla guarantees," in *Proceedings of the 28th*

ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, New York, NY, USA, Dec. 2022, pp. 62–77.

[35] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, Virtual Event Germany, Oct. 2021, pp. 724–739.

[36] W. Chen, Z. Mo, H. Xu, K. Ye, and C. Xu, "Interference-aware multiplexing for deep learning in gpu clusters: A middleware approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, Nov. 2023, pp. 1–15.

[37] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus OH USA, Oct. 2019, pp. 1063–1075.

[38] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," Dec. 2022.

[39] S. Zagoruyko and N. Komodakis, "Wide residual networks," Jun. 2017.

[40] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," Jul. 2023.

[41] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, New York, NY, USA, Oct. 2023, pp. 611–626.

[42] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, and X. Liu, "Elasticflow: An elastic serverless training platform for distributed deep learning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, 2023, pp. 266–280.

[43] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards automated slos for enterprise clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, USA, Nov. 2016, pp. 117–134.

[44] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on gpu clusters for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, 2020, pp. 533–548.