Proc. of International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA 2026)
5-7 February 2026, Boracay Island-Philippines

2026 International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA) | 979-8-3315-7191-7/26/$31.00 ©2026 IEEE | DOI: 10.1109/ACDSA67686.2026.11468130

# Optimizing Machine Learning Inference Serving Systems: A Survey

Shirin Ebadi
*Department of Electrical, Computer and Energy Engineering*
*University of Colorado Boulder*
Boulder, USA
shirin.ebadi@colorado.edu

Eric Keller
*Department of Electrical, Computer and Energy Engineering*
*University of Colorado Boulder*
Boulder, USA
eric.keller@colorado.edu

***Abstract*—For a long time, Machine Learning (ML) training has been the primary focus of both academia and industry, while inference was often treated as a simpler process that developers manually assemble its components. However, the demand for inference services has been rapidly increasing, with ML inference consuming more than 90% of infrastructure costs at Amazon Web Services. This makes the development of efficient inference systems increasingly important. The performance of these services, beyond model-level optimizations, is highly dependent on system-level optimizations, where various trade-offs arise. This survey examines techniques for resource usage optimization and latency improvement in ML inference serving within server-based infrastructures, focusing on research contributions from the last three years. We explore key optimization areas, including resource scheduling and autoscaling methods, batching and throughput improvement techniques, multi-tenancy and GPU sharing strategies that mitigate interference, latency–accuracy trade-off mechanisms such as model selection, fast-path optimizations and early-exit strategies, as well as emerging paradigms like programmable data planes and serverless computing. By summarizing recent works from top venues and highlighting open challenges, we aim to provide a comprehensive overview of the latest advances, opportunities, and future research directions in ML inference serving system optimizations.**

***Keywords*—Machine Learning, Inference System, Resource Management, Latency Improvement***

## I. Introduction

With the growth of Machine Learning (ML) usage across sectors such as advertisements [1], healthcare [2]–[4], and finance [5], [6], these systems receive an ever-increasing number of requests to process and respond to. As request volumes continue to grow, companies must dedicate increasing resources to serving ML models. At Amazon Web Services, for example, ML inference consumes more than 90% of infrastructure costs [7]. Given this escalating demand, both industry and academia are constantly seeking ways to reduce costs through efficient resource utilization while still meeting user service level agreements (SLAs), such as specific accuracy and latency requirements.

In most cases, improving latency is achieved by serving requests on the nearest model or one with fewer current jobs to process. However, this approach comes at the cost of sacrificing accuracy and sometimes slightly violating SLA requirements. Conversely, improving throughput and resource usage typically imposes additional waiting time on requests, which can increase overall latency. Therefore, enhancing the efficiency of inference serving systems requires balancing three interconnected requirements: resource usage, latency, and accuracy. The boundaries that define acceptable limits for latency and accuracy are established within SLAs.

Looking at similar surveys, there are some works that target Deep Learning inference serving for multi-tenant or large-scale systems [8], [9]. However, they either focus solely on optimizing one configuration, such as multi-tenancy, or they primarily organize the landscape around instance-device mapping paradigms (e.g., single-instance single-device vs. multi-instance multi-device) rather than practical optimization objectives. Other surveys have concentrated on ML serving for IoT [10], data planes [11], or serverless computing [12]. Two main shortcomings exist in the available literature. First, they lack a practical categorization of optimization objectives and how they interconnect. Second, existing surveys concentrate on specific computing paradigms while not addressing server-based inference system mechanisms commonly deployed in production. Additionally, there are surveys that focus exclusively on LLMs [8].

In order to overcome the shortcomings in the current papers, in this survey, we examine resource usage optimization and latency improvement techniques from a system-level perspective while preserving accuracy in ML inference serving within server-based infrastructures. We focus on research contributions from the last three years to capture the most recent advancements and emerging trends in the field. Most of the works discussed in this survey, follow multiple goals and leverage multiple optimizations in their systems. However, in each section we specifically discuss one related optimization in the system. Table I presents the papers discussed in this survey along with their goals and their techniques for optimization.

This work was supported in part by the National Science Foundation (NSF), through awards 2326835 and 2241818.

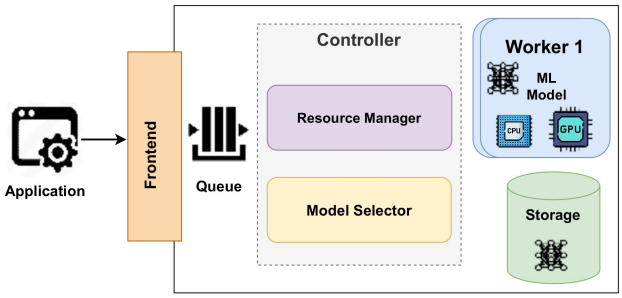


Fig. 1. A general architecture of an inference system.

By summarizing the existing works, we aim to provide a comprehensive survey on emerging challenges, opportunities, and innovations, and thus motivate new innovations in ML inference serving optimizations. This survey makes the following contributions:

- We provide a practical, goal-oriented taxonomy that categorizes inference serving optimizations based on interconnected objectives, such as resource usage and latency.
- We offer an in-depth analysis of server-based inference system mechanisms commonly deployed in production, covering recent advances (2022-2025) in ML inference serving.
- We identify open research directions, including heterogeneous accelerator scheduling and sophisticated caching strategies, to guide future work in this rapidly evolving field.

The rest of the survey is organized as follows: Section II describes inference serving systems. Section III presents resource usage optimization techniques. Section IV explores different methods used to improve latency in ML inference systems. Section V talks about two emerging paradigms for serving inference. Section VI discusses the future directions and open research problems. Section VII concludes this work.

## II. ML Inference System

Inference serving systems are specialized platforms designed to deliver accurate predictions for application queries while meeting agreed-upon latency requirements and minimizing infrastructure costs. Prior to the development of inference systems, developers were responsible for managing different models and frameworks for inference while ensuring scalability, accuracy, and optimization. However, as these systems have matured and multiple models with versioning are being served concurrently, managing them efficiently for each application has become increasingly complex. To address these challenges, numerous research efforts and studies are attempting to offer unified frameworks for serving ML workloads with multiple optimizations as shown in Table I.

A general architecture of these systems is showcased in Figure 1. The process initiates by applications submitting their query requests to a frontend, where they are queued in a central queue. The controller then dispatches queries from the queue and schedules them to workers. The resource manager and model selector work together to balance latency, accuracy, and cost. The resource manager allocates resources by setting up workers for ML inference and keeping models loaded on each worker. With limited workers and models available, the model selector chooses which models and workers should handle each query to meet the required latency and accuracy SLA.

Workers are compute resources that host ML models and serve batches of inference requests. They are typically a heterogeneous combination of CPUs, GPUs, and accelerators such as TPUs [24] or NPUs [25]. While GPU-based systems offer low latency for ML inference, achieving high utilization remains a challenging task, unlike ML training. The key difference between training and inference in terms of GPU utilization lies in their suitability for batching. During training, since input data is readily available, the system can batch large numbers of data samples, allowing GPUs to effectively leverage massive parallelism. In contrast, ML inference servers underutilize GPUs because they operate as on-demand systems where inference tasks are assigned to compute engines only after requests arrive.

## III. Resource Usage Optimization

This section examines proposed optimization methods for managing resources in ML workloads. As models and datasets grow in size, deciding where to place serving requests and models becomes increasingly critical. Research papers address this challenge from various perspectives.

*1) Resource Autoscaling:* Since ML workloads are bursty and fluctuating, it becomes important to scale the system efficiently based on the workload.

IRIS [13] considers the problem of scheduling in model-agnostic environments, where the scheduling framework is not designed for one specific model, but rather works for general ML models. They observed that preserving Quality of Service (QoS) by co-locating models on the same hardware resource can cause interference and degrade performance while optimizing resource usage. Based on their experiments, interference can degrade performance by more than 10× under heavy resource contention, with CPU and memory bandwidth stress causing the most severe degradation across different MLPerf engines. Regarding vertical scaling, they observed that increasing the number of threads (concurrency) has a linear correlation with performance improvement, while adding more workers (horizontal scaling) doesn't show significant improvement. In their experiments, they observed that the impact of interference or scaling can vary across different backends, which adds a challenge for model-agnostic schedulers. Considering this, they generally conclude from their experiments that optimal thread configuration is critical for maintaining performance under interference, with ONNX backends showing the best results at lower worker counts (workers = 1) and moderate thread counts (threads = 2-4), while TensorFlow backends achieve peak performance with higher thread counts (>5 threads). Notably, TensorFlow engines exhibit greater robustness to memory capacity interference compared to other

TABLE I. Hierarchical Classification of Discussed Papers in This Survey

| Category | Sub-category | Ref. | Optimizations | Evaluated Domains |
|---|---|---|---|---|
| Resource Usage Optimizations | Resource Autoscaling | IRIS [13] | ML Techniques | Image Classification |
| | Batching and Throughput Improvement | Proteus [14] | Accuracy Scaling, Adaptive Batching | Object Detection, Image Classification, NLP |
| | Multi-tenancy and GPU Sharing | Usher [15] | Multiplexing, Holistic resource optimization | Object Detection, Image Classification, NLP |
| | | GPUlet [16] | Adaptive Batching, GPU multiplexing | Object Detection, Image Classification |
| Latency Improvement | Fast Path | Autoscratch [17] | Caching, ML Techniques | Object Detection, Image Classification, NLP, DLRM, Speech Recognition |
| | | Cascade [18] | Zero-copy, Kernel Bypass | Object Detection, Image Classification |
| | Request Scheduling and Model Selection | RAMSIS [19] | Adaptive Batching, Stochastic Query Inter-arrival | Image and Text Classification |
| | | Loki [20] | Hardware and Accuracy Scaling, Early Dropping | Object/Facial Detection, Image Classification, Caption Generation |
| | Neural Architecture Early Exit | Apparate [21] | Automatic Early Exit, Continual Feedback | Object Detection, Image Classification, NLP |
| Emerging Techniques | Serverless Computing | Pisel [22] | Layer-grouping, Parallel Processing | Object Detection, Image Classification, NLP |
| | Data-Plane | Homunculus [23] | Bayesian-Optimization Design Space Exploration | Network Functions |

resource stressors, suggesting that careful tuning of parallelization parameters can significantly mitigate the impact of resource contention in shared ML inference environments

IRIS scheduling framework consists of offline and online phases. The offline phase of IRIS prepares training data and selects optimal ML models for performance prediction. It consists of two main stages that establish the foundation for runtime scheduling decisions. The first stage generates comprehensive training data through systematic scenario creation. IRIS deploys 1-16 iBench interference micro-benchmarks that stress different shared resources (CPU, cache, memory) while randomly selecting inference engines and parallelism configurations. Over 5,500 scenarios are executed, with continuous monitoring capturing both system-level metrics (IPC, cache misses, CPU utilization) and application performance (QPS) at 1-second intervals. These data are aggregated into a structured dataset that contains inference engine details, system metrics, parallelism settings, and achieved performance. The second stage performs ML model selection through design space exploration, evaluating multiple regression algorithms (Lasso, Random Forest, XGBoost, etc.) using 10-fold cross-validation. XGBoost consistently achieves the highest accuracy across inference engines and is selected as the primary prediction model. Hyperparameter optimization using randomized search further improves accuracy by an average of 1.5%, resulting in trained models capable of accurately predicting QPS performance under various interference conditions for use in the online scheduling phase.

The online phase of IRIS continuously monitors system interference and dynamically adjusts resource allocations to meet QoS requirements while minimizing CPU utilization. Model-agnostic configuration only requires the task type (image classification or object detection) and target QoS. The scheduler evaluates all available inference engines and parallelism levels for the given task and selects the combination that achieves the required performance with minimal resource consumption. Comparing model-agnostic against model-specific versions for the same tasks, for image classification, IRIS achieved 57.63% fewer QoS violations on average while maintaining 21.34% CPU utilization across all QoS constraints. For object detection, it achieved 23.89% fewer violations for low QoS constraints and performed similarly to the best model-specific scheduler for medium/high constraints while consuming only 34.19% CPU on average.

*2) Batching and Throughput Improvement:* It has been studied that GPU usage is more efficient when models process batches of inputs rather than single inputs. Batching can improve throughput. However, the critical challenge is finding the optimal batch size that achieves optimum resource usage while preserving SLA latency. Improper batch sizing creates performance trade-offs. Large batch sizes can violate latency SLAs for individual requests, while small batch sizes lead to queuing overhead and suboptimal throughput utilization. Batching can have a significant effect on the system; therefore, most of the works studied in this survey propose their own algorithms for determining the batch size. Here, we discuss one of the recent works.

Proteus [14] is an inference-serving system that handles varied query workloads by employing accuracy scaling. It manages resources by solving the problem using a mixed integer linear programming (MILP) framework when it detects changes in query demands. Given various query demand rates, it is crucial to design an online adaptive batching algorithm that adjusts batch size according to the number of queries arriving per second. To address this, Proteus proposes a non-work-conserving algorithm that waits as long as possible to accumulate more queries before executing a batch. The algorithm works by calculating the maximum time it can safely

wait for additional queries to arrive without causing the first query in the queue to violate its latency deadline. Specifically, for a queue with q queries, it computes $T_{max\_wait}(q+1) = T_{exp}(1) - T_{process}(q+1)$, where $T_{exp}(1)$ is when the first query expires and $T_{process}(q+1)$ is the processing time for a batch of $q+1$ queries. The system then waits until this deadline to accumulate more queries for larger, more efficient batches, and repeats this process iteratively to maximize batch size while ensuring no timeouts occur. They show that their algorithm achieves overall better performance compared to previous systems like Clipper [26] and InfaaS [27] with 2-4x fewer SLA violations.

*3) Multi-tenancy and GPU sharing:* By running models with mixed workloads on GPUs, concerns regarding fully utilizing GPUs and responding to complex multi-level queries of applications can be solved. Multi-tenancy has long been studied from model scheduling [28] and resource management [16] perspectives. Here we go through two scheduling frameworks for improving GPU usage in multi-tenant environments.

Usher [15] conducts a thorough experiment on existing systems and observes that current systems cannot simultaneously maximize computation and memory utilization, with spatial multiplexing causing up to 50% goodput reduction due to inter-model interference. They discover that optimal resource utilization requires holistic workload distribution across multiple GPUs (rather than independent per-model decisions), strategic pairing of computation-heavy models with memory-heavy models, and that model resource characteristics depend on batch size and latency constraints rather than just parameter count.

Usher maximizes GPU usage by addressing interference at three levels: resource allocation, memory management, and cache utilization. It has three main components: (i) Kernel-based Resource Requirement Estimator: A cost-optimized estimator that estimates resource usage for models with different configurations by converting high-level operator graphs into low-level GPU kernel execution graphs, then using regression models to predict when kernels will execute concurrently. (ii) Interference-aware and Resource Utilization Maximizing Scheduler: A two-phase scheduler that first groups models using k-means clustering to maximize opportunities for pairing computation-heavy models with memory-heavy models, aiming for balanced resource utilization where the sum of computation requirements approximately equals the sum of memory requirements within each group. Then, it performs holistic scheduling that considers all possible configurations of batch sizes and replication degrees for all models simultaneously. A key innovation is that the system may replicate a model across multiple GPUs even when one GPU could handle the entire workload to enable better overall cluster utilization through optimal multiplexing opportunities. (iii) Operator Graph Merging: This leverages the observation that models of similar architectures share significant weight similarities (52% of CNN pairs and 70% of Transformer pairs show substantial overlap). It first groups models with similar architectures using DBSCAN clustering, then creates bipartite graphs to match operators with similar execution characteristics and weight patterns. The merger extracts the largest common weight submatrices between matched operators and creates Super-Operators that execute shared computations simultaneously while the common weights remain in cache. They compared their scheduler against state-of-the-art systems, and reported that Usher can achieve up to 2.6× higher goodput and 3.5× better cost-efficiency.

GPULet [16] presents a spatio-temporal scheduling framework for serving heterogeneous machine learning models on multi-GPU servers. They observed that temporal sharing of GPU underutilizes GPU when a model doesn't use the whole resource for a task. On the other hand, sharing GPU spatially can downgrade the performance of inference when interference occurs. To solve this, they propose a GPU scheduling framework that assigns requests to an abstraction of GPU partition, called gpulet. Through their suggested elastic partitioning algorithm, they try to schedule an ML task across efficient set of gpulets. The algorithm begins by sorting models in ascending order by their request rate × SLA product to prioritize less demanding tasks first, then determines the optimal partition size by calculating $p_{eff}$ (maximum efficient partition representing the performance "knee point" during offline profiling) and $p_{req}$ (minimum required partition for SLA compliance in online phase), choosing $p_{ideal} = MIN(p_{eff}, p_{req})$ to avoid over-provisioning while meeting performance requirements. Using a greedy best-fit search strategy, it sorts remaining gpulets by size and allocates the smallest suitable partition, splitting unpartitioned gpulets when necessary, which reduces search complexity from $O(P^N NPM^2)$ to $O(NPM^2)$. For temporal sharing, they examine wether multiple gpulets can consolidate onto a single physical partition through time-multiplexing by caculating whether both models can meet their SLAs when sharing the larger of the two partition sizes. The algorithm chooses the larger partition and readjusts the batch sie and duty cycle to meet SLAs. The algorithm determines the maximum batch size while incorporating an interference prediction model based on L2 cache utilization and memory bandwidth consumption to account for performance degradation from concurrent model execution.

## IV. Latency-accuracy Tradeoff

Many queries are subject to SLAs that require responses within a specified timeframe. This temporal constraint often necessitates a tradeoff between response latency and accuracy. This section explores studies that address this challenge.

*1) Fast-path:* One way of improving latency without losing accuracy, is using fast-paths to return from the system earlier. Here we study caches and zero-copy as two fast-path methods.

GPUs have achieved significant Deep Learning (DL) performance improvement over the past year, e.g., NVIDIA's V100 GPU improved FP16 throughput by 6× compared to the previous generation, while NVIDIA's A100 GPU further increased it by 2.6× over the V100. However, GPUs' DRAM bandwidth scaling has not maintained similar growth and along with significant power consumption, systems tend to

reduce accesses to DRAM to save time and energy. While GPU manufacturers have responded by increasing L2 cache capacities, NVIDIA's Ampere generation GPUs have recently introduced L2 cache Residency Controls that allow programmers to selectively specify data as being L2 cache persistent. This mechanism effectively enables explicit data pinning in the GPU's L2 and protects it from being evicted by the default hardware-managed cache replacement policy. Leveraging this, AutoScratch [17] observed that activation data can be a good fit to reside in L2 cache. Unlike weights, activations are reused between layers, such that the activation output of a layer is used as an input to the next layer. Their size is proportional to the size of the batch, but as shown in their experiments, most layers' live activation data is small enough to reside in the cache while some layers may not, so they need to be selective about which activation slices to cache. Choosing optimal configurations from the billions of possible combinations requires expertise that most developers lack. The core innovation of AutoScratch lies in treating cache optimization as a machine learning problem. The system divides a GPU application's shared activation memory buffer into equal-sized slices (typically 1MB each) and uses ML algorithms to automatically determine which slices should be marked as L2-resident. AutoScratch offers two implementation variants with different performance characteristics.

AutoScratch-RL uses reinforcement learning with Proximal Policy Optimization, training directly on the target GPU hardware in an on-policy manner. While effective, this approach requires substantial training time averaging 330 minutes per workload. AutoScratch-EA employs a regularized evolutionary algorithm that initially maintains $M$ population of cache configurations and evolves them through mutation and selection in $k$ steps. This variant achieves similar performance benefits while reducing training time dramatically to just 5.7 minutes on average which is a 58× speedup over the RL approach. Their experimental results across the MLPerf [29] benchmark suit running on NVIDIA's L4 GPU showcases that system achieves a geometric mean reduction of 29% in DRAM traffic and 9% improvement in overall performance, with some workloads seeing up to 22% speedup.

Cascade [18] is a distributed edge ML hosting platform that is designed for serving latency-sensitive applications. It's core novelty lies in its RDMA-enabled KV store accompanied by interfaces that optimize communication. In addition, they follow the locality of data and computation paradigm and remove the data-copy overhead by running applications in the same address space of the Cascade.

Cascade allows user-defined lambdas (ranging from simple filters to complex AI models like YOLO) to execute in data flow graphs where each stage triggers the next, creating processing pipelines like image analysis chains for traffic cameras. The key innovation is that lambdas run either as Dynamic Link Libraries (DLLs) in Cascade's address space (receiving data as shared pointers) or in Docker containers connected via shared memory segments, ensuring data never gets copied between the storage and computation layers. The system employs multiple strategies to eliminate data copying: (1) it shares large memory-mapped objects between containers and Cascade without duplication and (2) assumes all servers use the same native data layout to avoid marshaling overhead during RDMA transfers. For object construction, Cascade allows applications to build data structures directly in pre-allocated, RDMA-ready memory buffers rather than constructing them elsewhere and copying. Clients provide constructor functions that Cascade calls when buffer space becomes available which enables objects like camera images to be built in-place and transmitted without any intermediate copying. This comprehensive approach to eliminating data movement at every stage, from GPU memory sharing to in-place object construction in RDMA buffers, ensures that the entire pipeline from data generation to network transmission avoids the performance penalties typically associated with distributed computing systems. Cascade achieves 3+ orders of magnitude latency improvements over existing platforms (100 microseconds vs milliseconds for messaging, 6ms vs 25ms for ML pipelines) while maintaining or exceeding throughput, demonstrating that its zero-copy RDMA architecture and edge-optimized design effectively solve the performance bottlenecks that plague current cloud-based streaming and AI platforms.

*2) Request Scheduling and Model Selection:* There is a line of works that tries to improve latency while considering accuracy by optimally selecting models and scheduling requests to be processed by them. Mendoza et al. propose RAMSIS [19] which is a model selection and scheduling framework that explicitly accounts for variable query load and stochastic inter-arrival patterns in order to achieve maximally accurate predictions for queries within a user-defined latency SLA. It consists of offline and online stages. In the offline stage, it profiles the inference latency for all models on all workers with all supported batch sizes and inference accuracy for each trained ML model. Considering the query arrival distribution, the probability of k queries arriving during a time interval of length T which captures the query load and the stochastic nature of the inter-arrival pattern, along with profiles and query latency SLA, it generates policies that are each specialized for a particular query inter-arrival pattern and response latency SLA for each worker. While most systems assume Poisson arrival distribution, RAMSIS is designed to be more flexible. It can be parameterized by different arrival distributions like Gamma distribution. During inference serving, which is the online phase, when inference queries arrive at the central queue, the load balancer distributes them across worker queues using a round-robin strategy to encourage high resource utilization. Meanwhile, a load monitor tracks the current query arrival rate using a moving average over a 500-millisecond window. For model selection, each worker has a dedicated model selector service that chooses the appropriate pre-computed policy based on the observed query load (it decides the lowest-load policy that can handle the anticipated load) and then redirects $b$ queries to selected model $m$ with batch size $b$. The RAMSIS framework provides probabilistic guarantees on accuracy and latency

SLA compliance, meaning that it provides the scheduler and user an expected fraction of all served queries which do not meet their deadlines. Comparing RAMSIS with state-of-the-art works like Jellyfish [30] and ModelSwitching [31] which only consider query load and are inter-arrival pattern agnostic, RAMSIS achieves up to 15% higher accuracy for image classification. Also, RAMSIS can achieve the same accuracy as the baselines with significantly fewer resources. In one experiment, they showed that RAMSIS requires 75% fewer resources than Jellyfish for text classification.

Loki [20] is another scheduling framework designed for inference pipelines. In the current inference ecosystem, most applications do not use a single model for performing a complex task; instead, they break the task into multiple sub-tasks that are handled by different models in a pipeline manner, e.g., first, one model detects objects in an image, then the next model detects fraud. Scaling these workloads by taking a pipeline-agnostic perspective that treats ML models independently without considering inter-model dependencies leads to suboptimal resource allocation and high SLA violation rates.

Loki [20] takes a combined accuracy-hardware scaling approach to solve this problem. It first tries to serve the incoming requests with the minimum number of resources, and when it determines that it is unable to meet demand by using the entire cluster with the most accurate model variants, it tries to determine the minimum amount of system accuracy to sacrifice in order to meet the demand. To optimally scale accuracy and hardware, it solves a mixed-integer linear program (MILP) to plan the resource allocation. Loki addresses the interdependencies between models through its MILP formulation by modeling end-to-end accuracy across complete source-to-sink paths rather than individual model accuracies, tracking multiplicative factors that capture how one model's accuracy affects downstream workloads (e.g., more accurate object detection creates more classification work), and using an optimization objective that maximizes weighted system accuracy to strategically determine which models' accuracy to reduce for minimal overall pipeline impact. Loki's Resource Manager runs periodically (every 10 seconds) to optimize resource allocation, using exponentially weighted moving averages to estimate demand and tolerating 500ms solve times since it's not on the critical path. The MILP solutions determine both batch sizes for workers and latency budgets for tracking SLA compliance, while accounting for communication delays between servers and using worker-reported multiplicative factors (how many downstream requests each input generates) to accurately model pipeline dependencies and throughput requirements. Next, the centralized Load Balancer uses this plan along with the pipeline graph and the recent demand history to generate routing tables for each worker. The authors compare their work with Proteus [14] which solely does accuracy scaling and InferLine [32] that performs hardware scaling alone on two scenarios. Loki reduces SLA violations by 10×, and server cost by up to 2.6× compared to Proteus while providing higher overall accuracy. Compared to InferLine, Loki increases the capacity of the cluster by 2.5×.

*3) Neural Architecture Early-exit:* Apparate [21] represents the first system capable of automatically incorporating and managing early exits (EEs) for serving diverse machine learning models without requiring developer intervention or specialized expertise. The fundamental premise underlying this approach is that certain "easy" inputs may not necessitate the full predictive capacity of a model to produce accurate results. Rather, outputs for such inputs can be reliably predicted from values computed at intermediate layers. In these instances, bypassing subsequent model execution yields proportional reductions in both per-request latency and computational overhead. Consequently, the objective of EEs is to identify, on a per-input basis, the earliest layer at which an accurate response can be generated. To implement EEs, intermediate layers within a model are augmented with computational ramps. These ramps process the output values from their associated layers and analyze them to predict the model's final result, such as a classification label.

Apparate focuses solely on delivering latency reduction, and unlike traditional systems, results for successful exits are immediately released, but all inputs continue to run to the end of the model to gain direct and continual feedback on EE accuracy. This crucial feedback mechanism is used to power novel runtime monitoring and adaptation strategies, which are essential for coping with time-varying overhead and accuracy challenges that EEs can bring.

By providing developers with automatic EE injection, Apparate needs to make decision about where to inject ramps and how to train them. Apparate identifies suitable points for ramps by marking operators that are "cut vertices" in the ONNX graph, ensuring that ramps leverage all available data flows without making decisions on partial data. This strategy results in ramps typically placed between major blocks in models like ResNet and BERT, or at all layers in VGG models, to maximize coverage and adaptation options. Apparate specifically opts for shallow ramps, generally comprising the model's final fully-connected (fc) layer prepended with a lightweight pooling operation to reduce dimensionality of intermediate layers. This design choice supports fine-grained runtime adaptation and has been empirically observed to be sufficient for dynamic workloads, even outperforming more complex ramp designs in terms of latency savings. Finally, for independent ramp training, Apparate freezes the original model weights during preparation and trains all ramps in parallel and independently of each other. This process uses the original model's outputs directly as labels and prohibits early exiting during training, which ensures that ramps are trained regardless of the presence or behavior of upstream ramps, maintains the original model's behavior, and enables rapid parallel training. Their evaluation on comuter vision and NLP workloads show that, Apparate lowers median latencies by 10 - 91%.

## V. Emerging Techniques and Paradigms for Serving Inference

*1) Programmable data plane:* With the advance of programmable data planes, like SmartNICs, switches, and etc. they can now execute more complex operations and ML models directly in the network at line-rate. However, offloading a suitable model to data-planes requires a high knowledge in ML to choose the right model and tune its hyperprameters as well as hardware knowledge to program it on specific data-plane. Homunculus [23] solves this problem by automatically generating efficient data-plane ML pipeline for different network application use cases. Homunculus recieves the user objectives, data, and constraints through an alchemey frontend. Then, through its optimization core, it performs a design space search for model selection that maximizes the application objective while respecting the data-plane resources and network constraints. Next, the backend generator, generates the suitable codes for a specific hardware that Homunculus supports. After a predetermined number of optimization iterations, the best performing, constraint-compliant model is selected, and its data-plane platform code and binaries are generated. The optimization core formulates ML model generation as a multi-objective Bayesian optimization problem, using HyperMapper to efficiently search across hyperparameters (learning rates, network architectures), physical resources (compute/memory units, match-action tables), and network constraints (throughput, latency requirements). It runs multiple parallel candidate explorations across different algorithm families (DNNs, SVMs, K-means), with each iteration involving the Bayesian optimizer suggesting new configurations, training models with those parameters, evaluating against accuracy metrics and feasibility constraints, and feeding results back to guide future search which typically converging to high-performing, constraint-compliant models within 20-30 iterations. The backend code generator transforms these abstract models into executable hardware-specific implementations using template-based synthesis, maintaining libraries of parameterized code templates for common ML operations (dot products, activations, matrix multiplications) that are instantiated with optimization-determined parameters and compiled to platform-native code (Spatial HDL for Taurus switches, P4 for MAT-based switches like Tofino). The backend performs platform-specific optimizations to maximize hardware utilization. It creates efficient dataflow graphs for SIMD compute grids on Taurus, while managing data movement through double-buffered SRAM, or mapping algorithms to table lookups for MAT-based architectures and integrates with cycle-accurate simulators (SARA, Xilinx Vivado) for ground-truth feasibility validation that ensures generated implementations actually meet specified performance and resource constraints.

*2) Serverless Computing:* Serverless computing comes with significant benefits like offloading autoscaling, resource management, deployment, and maintenance to cloud providers. These benefits makes it attractive for ML workloads, where the need for scalability, cost efciency, and rapid deployment is paramount [12]. However, serving applications in Serverless frameworks has some challenges that have long been considered and mitigated in different studies [12]. PISeL [22] observed that general solutions suggested for mitigating cold start problem in serverless environments are not useful in DNN serving workloads. The reason is DNN application bootstrap (framework initialization, model download, deserialization, and loading) has become the dominant factor in overall cold start latency, especially as model sizes continue to grow. This problem is compounded by severe memory consumption spikes during model loading that can easily trigger out-of-memory (OOM) errors, forcing users to provision larger, more expensive containers. To solve these problems, the propose a framework-agnostic pipelining solution that could transparently overlap model download, deserialization, loading, and execution stages without requiring modifications to existing DNN applications, while simultaneously reducing both cold start latency and peak memory usage across different frameworks like PyTorch and TensorFlow.

They solve these through a layer-grouping pipelined architecture that partitions models into consecutive layer groups and overlaps model download, deserialization, loading, and execution stages. The system uses a greedy partitioning algorithm that groups layers into partitions sized at least RL × BW (request latency × bandwidth) to minimize synchronization overhead, while a multi-component pipeline (Partition Agent, Downloader, Loader, Executioner) enables parallel processing where inference execution begins on early partitions while later ones are still downloading. The design maintains framework transparency through function hooks rather than deep modifications, uses condition locks for lightweight synchronization that's removed after full loading, and significantly reduces memory consumption by avoiding simultaneous storage of serialized and deserialized model data. PISeL achieves substantial performance improvements: 37-63% reduction in cold start times across PyTorch (CPU/GPU) and 29-33% across TensorFlow (CPU/GPU), while reducing peak memory usage by up to 59% for PyTorch and 30% for TensorFlow, enabling larger models to run on memory-constrained serverless environments without requiring application code changes.

## VI. Future Directions

### A. State-of-the-art Accelerators

While works such as Usher [15] and gpulet [16] demonstrate promising results in optimal scheduling and GPU sharing across diverse workloads, the ecosystem is rapidly evolving to incorporate other accelerators like NPUs and TPUs, which remain relatively unexplored. By understanding the fundamental differences between GPUs and NPUs, interesting research opportunities can be pursued in two layers: (i) At the compiler layer, a compiler can effectively compile models to offload computational segments to CPU, GPU, and NPU resources. (ii) At the scheduling layer, a scheduler can effectively orchestrate workloads and co-locate models across diverse resources, including NPUs. When scheduling heterogeneous

workloads, energy consumption per worker represents a critical consideration. Beyond model characteristics, the workers' underlying hardware composition significantly impacts energy consumption patterns. For instance, NPUs typically exhibit lower power consumption compared to GPUs [33]. Designing schedulers and systems that incorporate these energy metrics to optimize for energy-constrained applications remains largely unexplored.

### B. Sophisticated Caching

Caching can significantly enhance both latency and resource utilization; however, existing works mostly utilize simple caching policies. Caching can be examined from two perspectives: (i) determining which ML domains are most suitable for caching, for example, caching text may not provide substantial system gains compared to caching images, and (ii) establishing optimal cache policies customized for ML workloads, including how similarities should be calculated and which eviction strategies should be employed. One promising research direction involves exploring in-place caching updates. This update process is necessary to maintain accuracy as trends change and content is updated (e.g., new content becomes available). Naive, version-based updates, e.g., commissioning a new version of the model and decommissioning the old version can cause inefficiencies. It is possible to deploy a model at version $n + 1$ while version $n$ is still deployed; once $n + 1$ is fully available in memory, inference requests can be routed there instead. However, during the transition this would require two-fold the size of the model in memory utilization. Future research should investigate atomic in-place update techniques that can address these limitations while maintaining system reliability.

## VII. Conclusion

In this survey, we examined resource optimization and latency improvement techniques for ML inference serving from a system-level perspective. We identified three key areas: resource scheduling and management for maximizing GPU utilization, latency-accuracy tradeoffs including fast-path and early exit strategies, and emerging paradigms like programmable data planes and serverless computing. As inference demands grow, future research should focus on adaptive systems that balance efficiency, latency, and accuracy across heterogeneous hardware while remaining scalable and cost-effective.

## Acknowledgment

This work was supported in part by the National Science Foundation (NSF), through awards 2326835 and 2241818.

## References

[1] Y. Chen and J. Liu, "Ai-driven marketing strategy optimization in the digital economy: A machine learning approach," in *Proceedings of the 2025 International Conference on Digital Economy and Intelligent Computing*, ser. DEIC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 239–242. [Online]. Available: https://doi.org/10.1145/3746972.3747010

[2] S. A. Ajagbe and M. O. Adigun, "Deep learning techniques for detection and prediction of pandemic diseases: A systematic literature review," *Multimedia Tools and Applications*, vol. 83, no. 2, pp. 5893–5927, 2023.

[3] A. Kamankesh, N. Rahimi, I. G. Amiridis, C. Sahinis, V. Hatzitaki, and R. M. Enoka, "Distinguishing the activity of flexor digitorum brevis and soleus across standing postures with deep learning models," *Gait Posture*, vol. 117, pp. 58–64, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0966636224007495

[4] N. Rahimi, A. Kamankesh, I. G. Amiridis *et al.*, "Distinguishing among standing postures with machine learning-based classification algorithms," *Experimental Brain Research*, vol. 243, no. 3, 2025. [Online]. Available: https://doi.org/10.1007/s00221-024-06959-9

[5] J. Wang, "Big data financial fraud detection with machine learning," in *2024 IEEE 2nd International Conference on Electrical, Automation and Computer Engineering (ICEACE)*, 2024, pp. 599–603.

[6] Z. Liu, "Application of machine learning in financial risk classification and account verification optimization strategy," *Economics and Management Innovation*, vol. 2, no. 2, p. 64–70, Apr. 2025. [Online]. Available: https://www.gbspress.com/index.php/EMI/article/view/227

[7] Amazon Web Services, "Amazon EC2 Inf1 instances," https://aws.amazon.com/ec2/instance-types/inf1/, 2025, accessed: 2025-09-19.

[8] F. Yu, D. Wang, L. Shangguan, M. Zhang, X. Tang, C. Liu, and X. Chen, "A survey of large-scale deep learning serving system optimization: Challenges and opportunities," 2022. [Online]. Available: https://arxiv.org/abs/2111.14247

[9] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, and X. Chen, "A survey of multi-tenant deep learning inference on gpu," 2022. [Online]. Available: https://arxiv.org/abs/2203.09040

[10] K. Arikumar, S. B. Prathiba, R. S. Moorthy, K. Tamilarasi, M. V. Chalapathi, and A. Deepak Kumar, "The role of machine learning in iot: A survey," in *2022 3rd International Conference on Smart Electronics and Communication (ICOSEC)*, 2022, pp. 451–457.

[11] R. Parizotto, B. L. Coelho, D. C. Nunes, I. Haque, and A. Schaeffer-Filho, "Offloading machine learning to programmable data planes: A systematic survey," *ACM Comput. Surv.*, vol. 56, no. 1, Aug. 2023. [Online]. Available: https://doi.org/10.1145/3605153

[12] A. Aslani and M. Ghobaei-Arani, "Machine learning inference serving models in serverless computing: a survey," *Computing*, vol. 107, no. 1, Jan. 2025. [Online]. Available: https://doi.org/10.1007/s00607-024-01377-9

[13] A. Ferikoglou, P. Chrysomeris, A. Tzenetopoulos, M. Katsaragakis, D. Masouros, and D. Soudris, "Iris: Interference and resource aware predictive orchestration for ml inference serving," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, 2023, pp. 1–12.

[14] S. Ahmad, H. Guan, B. D. Friedman, T. Williams, R. K. Sitaraman, and T. Woo, "Proteus: A high-throughput inference-serving system with accuracy scaling," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 318–334. [Online]. Available: https://doi.org/10.1145/3617232.3624849

[15] S. S. Shubha, H. Shen, and A. Iyer, "Usher: holistic interference avoidance for resource optimized ml inference," in *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'24. USA: USENIX Association, 2024.

[16] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 199–216. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[17] Y. Fu, E. Bolotin, A. Jaleel, G. Dalal, S. Mannor, J. Subag, N. Korem, M. Behar, and D. Nellans, "Autoscratch: Ml-optimized cache management for inference-oriented gpus," in *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen, Eds. Curan, pp. 495–512.

[18] W. Song, T. Garrett, Y. Yang, M. Liu, E. Tremel, L. Rosa, A. Merlina, R. Vitenberg, and K. Birman, "Cascade: A platform for delay-sensitive edge intelligence," 2023. [Online]. Available: https://arxiv.org/abs/2311.17329

[19] D. Mendoza, F. Romero, and C. Trippel, "Model selection for latency-critical inference serving," in *Proceedings of the Nineteenth European*

*Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1016–1038. [Online]. Available: https://doi.org/10.1145/3627703.3629565

[20] S. Ahmad, H. Guan, and R. K. Sitaraman, "Loki: A system for serving ml inference pipelines with hardware and accuracy scaling," in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 267–280. [Online]. Available: https://doi.org/10.1145/3625549.3658688

[21] Y. Dai, R. Pan, A. Iyer, K. Li, and R. Netravali, "Apparate: Rethinking early exits to tame latency-throughput tensions in ml serving," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 607–623. [Online]. Available: https://doi.org/10.1145/3694715.3695963

[22] M. Rahimi Jafari, J. Su, Y. Zhang, O. Wang, and W. Zhang, "Pisel: Pipelining dnn inference for serverless computing," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, ser. CIKM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1951–1960. [Online]. Available: https://doi.org/10.1145/3627673.3679824

[23] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, and K. Olukotun, "Homunculus: Auto-generating efficient data-plane ml pipelines for datacenter networks," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 329–342. [Online]. Available: https://doi.org/10.1145/3582016.3582022

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," 2017. [Online]. Available: https://arxiv.org/abs/1704.04760

[25] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 449–460.

[26] 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)*. Boston, MA: USENIX Association, Mar. 2017, pp. 613–627. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[27] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 397–411. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/romero

[28] M. Lakshminarasimhan, M. Hall, S. Williams, and O. Antepara, "Brickdl: Graph-level optimizations for dnns with fine-grained data blocking on gpus," in *Proceedings of the 53rd International Conference on Parallel Processing*, ser. ICPP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 576–586. [Online]. Available: https://doi.org/10.1145/3673038.3673046

[29] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2020. [Online]. Available: https://arxiv.org/abs/1911.02549

[30] V. Nigade, P. Bauszat, H. Bal, and L. Wang, "Jellyfish: Timely inference serving for dynamic edge networks," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 277–290.

[31] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg, "Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, Jul. 2020. [Online]. Available: https://www.usenix.org/conference/hotcloud20/presentation/zhang

[32] 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*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 477–491. [Online]. Available: https://doi.org/10.1145/3419111.3421285

[33] Y. Hong and D. Kim, "Performance and efficiency gains of npu-based servers over gpus for ai model inference," *Systems*, vol. 13, no. 9, 2025. [Online]. Available: https://www.mdpi.com/2079-8954/13/9/797