HAMS: High Availability for Distributed Machine Learning Service Graphs

Shixiong Zhao*[1], Xusheng Chen*[1], Cheng Wang*, Fanxin Li*, Ji Qi*, Heming Cui*[2], Cheng Li†, Sen Wang‡
*University of Hong Kong, †University of Science and Technology of China, ‡Huawei Technologies[3]
*{sxzhao, xschen, cwang2, fxli, heming}@cs.hku.hk, †{chengli7}@ustc.edu.cn, ‡{wangsen31}@huawei.com

***Abstract*—Mission-critical services often deploy multiple Machine Learning (ML) models in a distributed graph manner, where each model can be deployed on a distinct physical host. Practical fault tolerance for such ML service graphs should meet three crucial requirements: high availability (fast failover), low normal case performance overhead, and global consistency under non-determinism (e.g., threads in a GPU can do floating point additions in a random order). Unfortunately, despite much effort, existing fault tolerance systems, including those taking the primary-backup approach or the checkpoint-replay approach, cannot meet all these three requirements.**

To tackle this problem, we present HAMS, which starts from the primary-backup approach to replicate each stateful ML model, and we leverage the causal logging technique from the checkpoint-replay approach to eliminate the notorious stop-and-buffer delay in the primary-backup approach. Extensive evaluation on 25 ML models and six ML services shows that: (1) in normal case, HAMS achieved 0.5%-3.7% overhead on latency compared with bare metal; (2) HAMS took 116.12ms-254.19ms to recover one stateful model in all services, 155.1X-1067.9X faster than a relevant system Lineage Stash (LS); and (3) HAMS recovered these services with global consistency even when the GPU non-determinism exists, not supported by LS. HAMS's code is released on github.com/hku-systems/hams.

I. INTRODUCTION

Recent machine learning (ML) models are pervasively deployed in mission-critical services (e.g., autopilot [47] and online stock prediction [74]). An ML service works as a dataflow application with a directed service graph [10], [34]. Each vertex in the graph represents an operator (i.e., an ML model) deployed on a distinct host to harness heterogeneous hardware resources; each edge represents an ordered connection from an upstream operator to a downstream operator, and the upstream operator propagates its outputs as a sequence of input requests to the downstream. A client program sends a request to the service graph and receives the final output from the graph as a reply.

An ML operator can be stateless or stateful. A stateless operator (e.g., inference with a VGG19 model [73]) processes each request independently and does not keep any states; a stateful operator (e.g., an online learned VGG19 model [71] or a Recurrent Neural Network model [20]) holds an internal state computed from previous requests that will affect the processing of future requests. For instance, Figure 1 shows the service graph for an online learned VGG19 model that has two input sequences of training and prediction requests with non-deterministic interleaving. The VGG19 model is stateful because previous training requests updates its model parameters that will affect future inference results.

Given that failures can happen at any time on any host, fault tolerance is crucial for mission-critical ML service graphs. A practical fault tolerant system for an ML model service graph should meet three essential requirements. First, the system must provide high availability with at most sub-second recovery time to provide continuous and unaffected services (e.g., an autopilot service should act with at most sub-second delay [47], [60]). Second, the system should incur low performance overhead in the normal case (no failure) compared with a bare metal (not fault tolerant) system.

Third, when a stateful operator O's state is recovered after a failure, the system must recover O's state with global consistency [70], [72], [84], [86]: as O's state affects its prediction output, if some of O's outputs are already processed by downstream stateful operators (or clients), O should not be recovered to a state that generates conflicting outputs (e.g., an output with same sequence number but a different value). For instance, if an online learned VGG19 model (Figure 1) tells its downstream operators and clients that the 34^{th} image is a truck, after a failover, its recovered state should not classify this image to a different result.

Non-determinism is the major open challenge for ensuring global consistency, as it has two sources in an ML service graph. First (**S1**), an operator can receive non-deterministically interleaved input sequences from multiple upstream operators. Second (**S2**), GPU models' processing of a request is inherently non-deterministic: the GPU scheduler can process floating point additions of multiple GPU threads in a non-deterministic order, and floating point additions are non-associative and rounded [55].

Due to **S2**, after an operator recovers from a failure, even given the same input sequence as before the failure (i.e., **S1** is eliminated), the recovered operator can easily run into a different (inconsistent) state, easily leading to disasters for mission-critical services. We did an experiment to reveal the consequence of **S2** during a failover triggered by us in Figure 2: an online learned VGG19 image classification model generates conflicting outputs on processing the 34^{th} inference request, permanently corrupted the service logic. We inspected the classification confidence tuple in the model's output, and this tuple changed from `(truck:0.5953, cloud:0.5884)` before the failure to `(truck:0.5921, cloud:0.5943)` after the failover. A practical fault tolerant system that meets the third requirement should recover the model to a state that

[1]The first two authors contributed equally.
[2]Heming Cui is the corresponding author.
[3]Theory Lab, 2012 Labs, Huawei Technologies, Co. Ltd., Hong Kong.

Fig. 1: An image classification service with an online learned VGG19 model. Grey models are stateless, and the blue model (i.e., the online learned VGG19) is stateful.

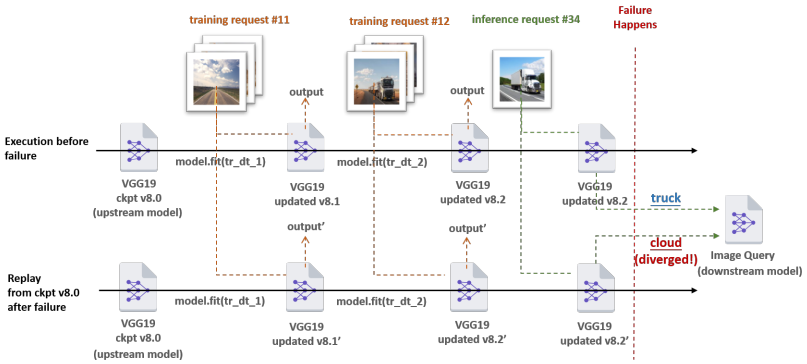


Fig. 2: Inconsistency caused by a failover using the checkpoint-replay approach for an online-learned VGG19 model. Its state is represented as Vx.y: Vx.0 means the state of the x^{th} checkpoint, and Vx.y means its state after processing y training requests since the x^{th} checkpoint. Inference requests do not change VGG19's state.

generates a consistent classification decision with the former tuple's, as this tuple has been passed to downstream operators or clients. We will illustrate GPU non-determinism in §II-C.

A recent system Lineage Stash [84] (LS) checkpoints the state of a stateful model periodically and uses causal logging [4] to efficiently record the interleaving of input sequences (**S1**). When failures occur, LS replays the interleaving from the latest checkpoint state. However, LS faces a paradox between the recovery time (the first requirement) and the normal case performance overhead (the second requirement) depending on its checkpoint interval. By default, LS sets its checkpoint interval to a few minutes, introducing a long recovery time of several minutes [84]. If LS reduces its checkpoint interval to sub-second for faster recovery, it incurs prohibitive latency slowdown (81% in our evaluation) because all stateful operators must be frequently stopped, checkpointed, and resumed. Moreover, LS, as well as other checkpoint-replay based fault tolerant systems [1], [6], [21], [50]–[52], assume **S2** does not occur, but non-determinism is inherent in GPU computing (§II-C). In short, the checkpoint-replay approach can meet either the first or the second requirement, but not the third one.

The primary-backup replication approach (e.g., Remus [13]) can meet the first and the third requirement (i.e., handling both **S1** and **S2**). It lets a primary execute a batch of requests, and propagate its updated state to a standby backup, so that the backup can take over immediately if the primary fails, meeting the first requirement. To meet the third requirement (global consistency), the primary buffers its outputs until the updated state is delivered to the backup. However, the primary-backup approach does not meet the second requirement due to two reasons. First, a primary of a stateful operator needs to be *stopped* to copy its updated state after processing every batch of requests. Second, the primary needs to buffer its outputs, which are demanded by its downstream operators, until the updated state is delivered to its backup. When multiple stateful operators are deployed in a service graph, each client request will be stop-and-buffered multiple times in the graph. We will illustrate the primary-backup approach in §III-B.

We present HAMS (Highly Available Machine-learning Services), the first ML service system that meets all the three crucial requirements. HAMS presents an ML-context-aware Non-Stop Primary-Backup protocol (NSPB) to eliminate the stop-and-buffer delay for an ML service graph. Our key observation is that the computation of a typical stateful ML operator can be divided into two phases: computation phase and update phase (§II-B). A stateful operator reads only its internal state in the computation phase and updates the state in the update phase. HAMS provides simple API for the ML model developers to explicitly identify the two phases, so that HAMS can asynchronously retrieve a model's updated state during the computation phase without stopping the operator.

Moreover, NSPB enables the primary to release its outputs to downstream operators without waiting the state to be delivered to the backup. Our idea is to let downstream operators *speculatively* execute these outputs, in parallel with the primary's delivery of its state to its backup. NSPB just maintains the causal dependency of per-batch state across the upstream and downstream operators. If any host (primary or backup) of any stateful operator fails, HAMS can maintain the leftover hosts' global consistency.

We implemented HAMS with about 11K LoC on Clipper [11] and Tensorflow Serving [58]. We evaluated HAMS with 25 mature ML models on PyTorch [65] and used these operators to build six practical ML services. We compared HAMS with a bare metal system, a HAMS-Remus system that follows Remus's primary-backup protocol, and an implementation of the most relevant replay-based approach LS [84], as LS is not open-source. Evaluation shows that:

- HAMS was efficient. HAMS's end-to-end latency achieved 0.5% to 3.7% overhead compared with the bare metal system, and this overhead is comparable to LS's.
- HAMS took 116.12ms-254.19ms to recover one stateful model in all services, 155.1X-1067.9X faster than LS.
- HAMS correctly recovered failed stateful ML operators in a service graph, in the presence of **S1** and **S2**.

The major contribution of this paper is HAMS, the first efficient and high available ML service graph system and the NSPB protocol. NSPB is a novel integration of the

primary-backup approach with the causal logging technique from the checkpoint-replay approach. HAMS's NPSB protocol can be integrated into diverse ML serving systems (e.g., Clipper [11], Tensorflow Serving [58], and Ray [50]) and improve their reliability.

In the remaining of this paper, §II introduces background. §III gives an overview of HAMS. §IV presents NSPB. §V describes HAMS's implementation. §VI shows our evaluation. §VII discusses related work and §VIII concludes.

II. Background and motivation

A. Machine Learning

Among all ML algorithms, we mainly focus on deep neural network algorithms [8], [26], [68], [73] in this paper as they are most widely deployed. The life-cycle of an ML model can be divided into two phases: a training phase and an inference phase. The training phase passes an input dataset multiple times (a.k.a., epochs), where each epoch works with four steps: (1) a portion (batch) of the input dataset forward-propagates through the model in parallel (forward propagation stage); (2) the model computes a loss for each input data (loss computation); (3) losses of the batch of inputs backward-propagate through the model in parallel with each leading to a gradient (backward propagation stage); and (4) the model is updated according to the sum of all gradients. In the inference phase, new (batched) inputs forward-propagate through the model to generate prediction results.

An ML model (or operator, interchangeable in this paper) can involve only the inference phase (model serving) or both the training and the inference phases (online learning). For model serving, a pre-trained model is deployed to give predictions on input requests. For online learning, the model is continuously re-trained with real-world data to fine-tune the model and to meet real-world trends, and serves inference requests at the same time. HAMS supports both deployment scenarios. As an ML model typically focuses on doing a single job (e.g., audio transcribing or face recognition), and different models are developed by different people, an ML service needs to compose multiple models in a service graph [10], [34].

B. ML Services Can Be Stateful

Stateful ML services have two major categories: stateful inference and stateful online learning. In stateful inference, an operator typically contains a Recurrent Neural Network (e.g., LSTM [20]) to capture dependencies in the input sequence (e.g., speech translation) as the model's state that will affect future predictions. In stateful online learning, the model's parameters are continuously updated and constitute its state.

For both categories, processing a (batch of) request can be divided into two stages: *computation* and *update*. For stateful inference, we take LSTM with a forget gate [20] as an example. When a new input request arrives, an LSTM cell first computes the forget gate's activation tensor, the update gate's activation tensor, and the output gate's activation tensor. All these three computations *read only* the hidden state tensor (computation stage). After that, the cell state tensor is updated according to previous computation results, the hidden state tensor is updated according to both the computation results and the updated cell state tensor (update stage). For stateful online learning, only training requests update the model's state (i.e., model parameters). In the four steps of the training phase (§II-A), the model's parameters are *read-only* in the first three steps and are only updated in the last step.

C. ML Can Be Non-deterministic

ML mainly does parallel computation (e.g., convolution) on matrices of floating points. Floating point additions are inherently not associative on GPU. Specifically, $a + b + c$ is usually not equal to $a+(b+c)$ because floating point numbers are rounded [55], which will accumulate in stateful models. This makes ML computation non-deterministic: the scheduler in a general GPU usually non-deterministically schedules the parallel additions (e.g., `AtomicAdd()` in CuDNN) of floating point tensors and produces different results, even given the same inputs, same hyperparameters, and same random seeds.

For instance, three back propagation algorithms (e.g., `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` [54]) in CuDNN are causing non-determinism due to the non associative property of floating point additions and non-deterministic parallel scheduling of GPU [55]. These three algorithms are frequently used by convolution operations in many ML frameworks (e.g., PyTorch [61], Tensorflow [1], and Caffe2 [29]), and causes both training and inference of a convolutional layer non-deterministic. Figure 2 illustrates a non-deterministic problem during online-learning caused by these three algorithms. In the backward propagation phase of a CNN model, the `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` algorithm is used. These algorithms can also make ML inference (with only forward propagation) non-deterministic. For instance, in the forward propagation of a transposed convolution layer (used for upsampling, a.k.a, deconvolution layer), the three algorithms are also invoked (as shown in the source code of Pytorch [66], Tensorflow [76], and Caffe2 [5]) and hence the computation can be non-deterministic. Moreover, some ML runtime features (e.g., autotune [7], [81] and runtime fusion [27]) can easily cause non-determinism.

An ongoing project [56] from Nvidia tries to eliminate some of these non-determinism sources by providing a more deterministic but slower CuDNN backend. However, some operations have non-determinism sources inherently, which are difficult to eliminate. For example, in PyTorch [61], even enabling the deterministic option (`torch.backends.cudnn.deterministic`), a set of operations (e.g., `ctc_loss()` and `embedding_bag()`) using parallel `AtomicAdd()` are still non-deterministic [67].

These non-determinism sources make an operator replayed from a checkpoint easily run into an inconsistent (divergent) state once the operator fails, even given the same input sequence as before the failure (Figure 2). We further conducted a quantitative study (Figure 3) to show how often a divergence may happen during a checkpoint-replay failover. The experiment used a Kaggle [33] dataset and a mature

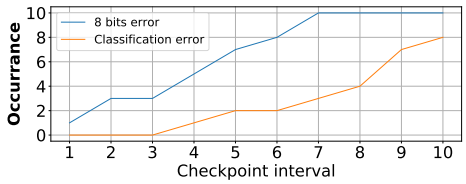


Fig. 3: Divergence occurrences of evaluating a model replayed from a checkpoint on a test set of 182 images (vehicle license number plates) by 10 times. X-axis is the checkpoint interval. Y-axis is the divergence occurrences.

Mask-RCNN [25] model built with `ctc_loss()` and enabled the PyTorch deterministic option. This model often exists in mission-critical autopilot systems for license plate recognition. In Figure 3, we counted the occurrences of classification errors whenever the recovered model gave an inconsistent (different) classification result on any of the tested image compared with that of the original model; we also counted the occurrences of 8-bit errors whenever the recovered model output an inconsistent rounded 8-bit precision loss on the whole test set compared with the original model. The results show that a longer checkpoint interval (§I) in the checkpoint-replay approach causes more occurrences of inconsistency during failover. Checkpoint-replay systems like LS [84] typically take a long checkpoint interval (e.g., one checkpoint per 100 requests) for efficiency, which makes divergence occur more often.

III. Overview

A. System Model and Architecture

Figure 4 shows HAMS's architecture. To deploy an ML service graph, a developer provides a definition of the graph and a set of pre-trained models. HAMS deploys these models on a cluster of physical hosts with each model co-located with a HAMS proxy. A HAMS proxy encapsulates the logic for request propagation, state replication, and failover, while a model only takes a batch of inputs and produces outputs. A developer should specify whether each model is stateful or stateless in the DAG. HAMS replicates each stateful model with a primary and a backup for fast failover. HAMS do not replicate stateless models as they do not hold internal state. HAMS also provides a group of frontend servers replicated with SMR [59] to handle client requests: on receiving a client request, a frontend logs the request, sends it to the service graph, and returns the result processed by the service graph back to the client. HAMS has a global manager replicated with SMR [59] for each deployment domain (e.g., a datacenter) to store deployment information and to handle failover.

HAMS supports general ML services represented as Directed Acyclic Graphs (DAG). Cyclic graphs with back-edges (e.g., reinforcement learning [31]) can be easily converted to DAGs in HAMS by letting their back-edges point to the frontend. In a DAG, if a model has multiple input sequences, the service developer can determine whether to let the model combine requests from these sequences or process them in an arbitrarily interleaving manner; if a model has multiple output streams, each output of the model can be sent to a specific subset or the entire set of downstream operators.

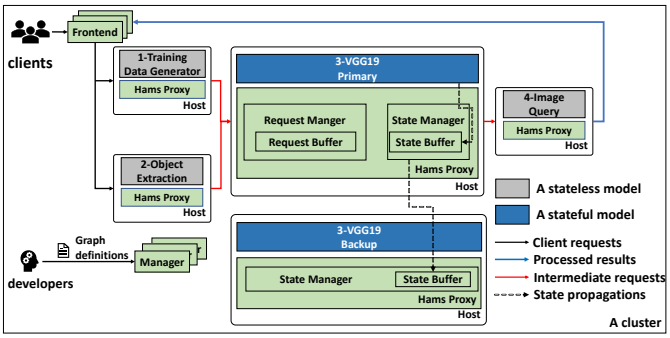


Fig. 4: HAMS architecture with components in green. A cluster can be a datacenter or a car with computers (e.g., autopilot).

As shown in Figure 4, a HAMS proxy for the primary of a stateful model has two main modules. The *Request Manager* buffers received input requests from upstream models, logs metadata for these requests (§IV-D), and synchronously passes a batch of requests to the model for processing. When the model finishes processing a batch of requests, the *State Manager* retrieves the model's internal state into a state buffer and sends the state to its backup.

A HAMS proxy for a stateless model activates only its request manager module; a HAMS proxy for the backup of a stateful model activates only its state manager module. When a backup's proxy receives a state from its primary, it first saves the state in the state buffer, and determines when to apply the state to the model based on HAMS's NSPB protocol (§IV-C).

Failure model. HAMS's failure model is the same as typical fault-tolerant systems for dataflow applications [84], [86]: a host can fail, network packets can be dropped or reordered, and network can be partitioned. As a primary-backup system, HAMS assumes that the primary and backup of the same model do not fail simultaneously: HAMS handles one host failure for each stateful operator or each stateless operator (§IV-E). HAMS also uses efficient state machine replication [59] to replicate the frontend as it is deterministic.

B. Context-aware Non-stop Primary-backup

Primary-backup is a powerful approach to provide fault tolerance for applications with internal non-determinism. For instance, Remus is a notable fault-tolerance system for generic applications running with a virtual machine (VM) as a black box. The upper half of Figure 5 shows the workflow of Remus. Remus first lets the primary execute a batch of requests (say the n^{th} batch) and buffers its outputs, and then replicates the primary's state with three steps. First, Remus stops the primary and copies its updated states to a memory buffer. Second, Remus lets the primary execute the $(n+1)^{th}$ batch of requests and sends the copied state to the backup. Third, once the backup successfully receives the state update, the primary releases its output for the n^{th} batch to downstream models. This output buffering is essential to ensure global consistency.

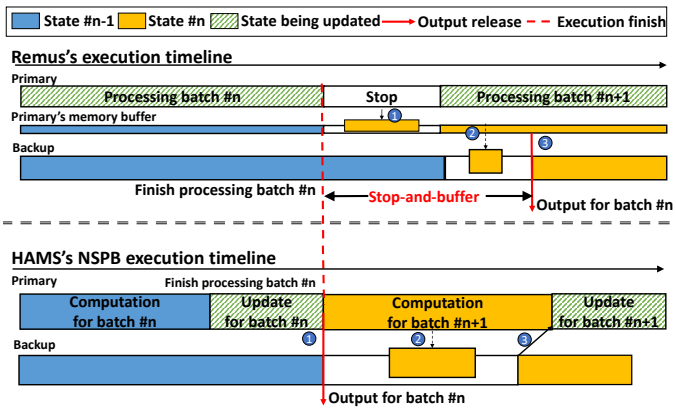


Fig. 5: Comparison of Remus and NSPB: NSPB releases outputs to downstream models much faster than Remus.

However, this traditional primary-backup approach is significantly inefficient for replicating stateful models in a service graph: the processing for the n^{th} batch finishes at the time of the red dotted line, but the output is buffed until the third step finishes. In a service graph, the delay caused by buffering will occur multiple times when a client request goes through multiple stateful models in the graph, causing prohibitive slowdown for mission-critical services.

The bottom half of Figure 5 shows the workflow of HAMS's NSPB protocol with three steps. First, the primary releases the output to downstream operators immediately after processing the n^{th} batch of requests. Second, the primary sends its state made by the n^{th} batch to its backup in parallel with the computation stage of the $(n+1)^{th}$ batch of requests. Third, the primary enters its update stage for the $(n+1)^{th}$ batch after the state made by the n^{th} batch is delivered to the backup. NSPB eliminates the stop-and-buffer delay in a traditional primary-backup system and incurs little performance overhead. We carry a proof of NSPB's global consistency in a service graph in §IV-F.

IV. HAMS's Runtime Protocol

A. Preliminaries

As a service's models work in a DAG, we can topologically sort the models and use M_p to denote the p^{th} model. We say that M_p is a *predecessor* of M_q if there is an edge $(M_p \rightarrow M_q)$ in the graph, and M_q is a *successor* of M_p. We say M_x is M_p's *downstream* model if M_p can reach M_x in the graph, but M_x is M_p's successor only if they are adjacent in the graph.

We use r_p^i to denote the i^{th} request that the M_p model executes, s_p^i to denote the resultant state ($s_p^i = \varnothing$ if M_p is stateless), and o_p^i to denote the corresponding output. We say a state s_p^i is *durable* [84] if M_p's backup has applied the state. To ease discussion, we further define a stateful model M_p's *next stateful models* as the nearest downstream stateful models for M_p. Formally, a stateful model M_y is an M_p's next stateful model if: there exists one path that M_p can reach M_y and there is no other stateful model in the path between M_p and M_y. Symmetrically, we define *previous stateful models* as the nearest upstream stateful models.

B. Retrieves A Model's State without Stopping It

Existing primary-backup approaches need to stop the primary to retrieve its internal state. For instance, Remus leverages VM-specific techniques (e.g., shadow page tables) to record updated memory pages (a.k.a, dirty pages) on the primary VM and periodically stops the primary VM for retrieving the dirty pages to send to the backup.

However, this black-box approach is not suitable for replicating stateful ML models in a service graph as recent ML models typically run on GPUs. The bandwidth between GPU memory and CPU memory is around one order of magnitude smaller than the bandwidth between CPU and its memory [62]. If Remus's black-box approach is used, the stop time will be much longer for retrieving dirty GPU memory. Moreover, it is still an open challenge [14], [16] to efficiently identify dirty GPU pages, so Remus has to copy all GPU memory out first.

Unlike Remus, NSPB introduces a white-box mechanism based on the compute-then-update nature of ML models (§II-B) to retrieve a stateful model's internal state without stopping it. As illustrated in Figure 5, during the computation stage for the $n+1^{th}$ batch, the primary model's model state (i.e., parameters) stays intact, same as after processing the n^{th} batch. Thus, the primary's proxy can retrieve the model's internal states in parallel with the computation stage of the $n+1^{th}$ batch, without the need to stop it.

Occasionally, state retrieval may take longer time than the computation stage. To prevent the proxy from getting corrupted state, HAMS lets the primary model wait for a signal from the proxy (§V) before entering the update stage, the proxy sends the signal after finishing its state retrieval.

C. Release Outputs without Waiting for State Propagation

For clear exposition of idea, in this subsection, we use a deliberately simplified setting with models working in a service chain (no branches) and batch sizes all being one.

To ensure global consistency, existing primary-backup approaches let a primary buffer a request's output until the state modified by this request is delivered to its backup (i.e., durable). Previous works [15], [46], [83] show that output buffering is a major source of overhead to primary-backup systems, and the overhead will increase with the replicated application's memory footprint.

NSPB enables a stateful model's primary to release outputs without waiting for its states to be durable. However, NSPB must maintain global consistency. A strawman approach is to buffer each reply to its client only before the reply leaves the service graph, until *all* states affected by this reply are durable. This approach works for deterministic services [36], but will cause inconsistency in the presence of non-determinism.

Figure 6 shows an example of the inconsistency caused by the strawman approach: model A (M_a) and model B (M_b) are two stateful models, and models between A and B are stateless. The inconsistency can be triggered with five steps. First, M_a's primary processes request r_a^n, generates its output o_a^n and state s_a^n. Second, M_a releases its output to downstream models and asynchronously sends s_a^n to its

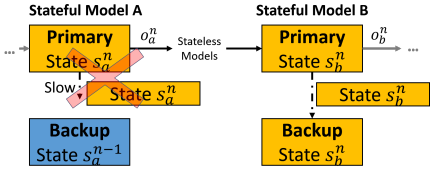


Fig. 6: Inconsistency caused by the strawman approach.

backup. Due to network anomaly, the delivery of s_a^n is delayed. Third, o_a^n is processed by stateless models between M_a and M_b and is transformed as r_b^n that is processed by M_b. Forth, M_b finishes processing r_b^n and successfully delivers the state s_b^n to its backup. Fifth, M_a's primary fails and its backup takes over as a new primary. As M_a's new primary still holds state s_a^{n-1}, the new primary needs to re-execute r_a^n (for now, assume HAMS already replicates input requests). However, as the re-computation is non-deterministic, M_a's backup may enter a different state $s_a^{n'}$ and generate a different $o_a^{n'}$, causing inconsistency (with the same consequence as Figure 2). Existing stream processing systems tackle this problem with a globally coordinated rollback [18], where all stateful models invoke rollbacks to the latest globally checkpointed state. However, such an approach causes a long recovery time [84] due to the expensive globally coordinated rollback, and a high normal case overhead because a reply can only be released to a client after a global checkpoint finishes.

To address this problem, we observe that the inconsistency is triggered because the failover breaks the causal dependencies of s_b^n on s_a^n, and global consistency can be maintained if HAMS can ensure the following statement: *s_b^n always becomes durable after all states that s_b^n depends on (i.e., s_a^n) are durable*. This can be achieved if we hold the state of s_b^n on M_b's backup until s_a^n is durable.

If the primary of M_a fails, HAMS promotes both M_a's and M_b's backups as their new primaries, and now the new primary of M_a is in state s_a^{n-1} and the new primary of M_b is in state s_b^{n-1}, with all their causal dependencies maintained. HAMS simply lets predecessor models of M_a resend its n^{th} output and lets the new primary of M_a and M_b re-process their n^{th} request. In a sense, HAMS regards each stateful model's execution as speculation before its state is durable, and HAMS can safely discard the speculative states on a failover.

One concern of this approach is how HAMS handles correlated failures if M_a's primary and M_b's backup fail simultaneously. HAMS handles this issue by letting M_b's primary free its buffer for the state s_b^{n-1} only *after* M_b's backup applies s_b^n (i.e., s_b^n is durable and s_b^{n-1} becomes outdated). Therefore, if such a correlated failure happens, M_b's primary can rollback to s_b^{n-1} by loading this state from its (CPU) memory buffer to GPU.

Note that HAMS retrieves a stateful model's complete state (e.g., all parameters for an online learned model or all cell states for an RNN) rather than state updates like Remus, so a rollback can be achieved as a simple state overwriting. If M_b's backup does not fail, however, HAMS first tries to promote M_b's backup as the new primary instead of to rollback its primary for fast failover, as rolling back the primary is slow

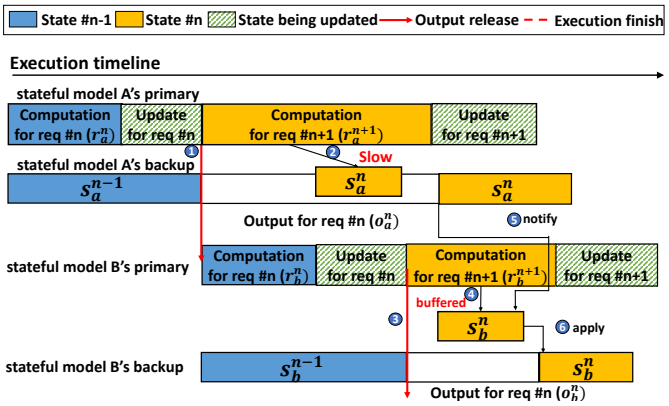


Fig. 7: HAMS's NSPB protocol in the simplified setting.

Algorithm 1: Each proxy records requests' lineage

```
1 my_seq ← counter of request sequence for the model
2 upon receiving req from pred_model with pred_seq
3     req.lineage.append
        ⟨pred_model, pred_seq, my_model, ++my_seq⟩
4     batch.atomic_append(req)
5 upon receiving outputs for reqs from the model
6     for i in range(reqs.size()) do
7         outputs[i].lineage = reqs[i].lineage
```

because it involves stopping its current GPU computation and copying the s_b^{n-1} from CPU memory to GPU, while the backup already has s_b^{n-1} loaded on GPU and can takeover immediately in HAMS (§VI-D).

Figure 7 illustrates NSPB in this simplified setting in six steps: (1) M_a's primary finishes processing r_a^n and sends output o_a^n to downstream; (2) M_a's primary sends s_a^n to its backup, which may be slow; (3) M_b's primary finishes executing request r_b^n and releases output; (4) M_b's primary sends s_b^n to its backup, and the backup put this state into a memory buffer; (5) M_a's backup notifies M_b's backup that s_a^n is durable; (6) M_b's backup applies state s_b^n.

D. NSPB for Complete Service Graphs with Batching

In HAMS, a stateful model's state is denoted as a three-tuple $\langle$`reqs`, `tensors`, `outputs`$\rangle$ and HAMS replicates the state after the model processes each *batch* of requests. `reqs` contains the requests' lineage information (explained in next paragraph) in this batch. `tensors` are the internal state of the stateful model, such as the value of the memory cells in a LSTM or parameter values for an online-learning model (§II-B). `Outputs` are the outputs from the model after processing this batch of requests. As a stateful model cannot re-execute its computation, HAMS saves this output in the memory to handle a downstream model failure. A saved output can be garbage-collected asynchronously when the client request of this output leaves the service graph.

For a general DAG with batching, we need to make two additional adaptations from the simplified protocol described in the previous subsection. First, to track causal dependencies for states in a general DAG, HAMS proxies need to maintain

Algorithm 2: A HAMS backup's algorithm

```
1  PFM[] ← Previous stateFul Models (see §IV-A)
2  NFM[] ← Next stateFul Models (see §IV-A)
3  durable_seqs ← a map for durable seq of each PFM
4  upon receiving⟨reqs,tensors,outputs⟩ from primary
5      for r in reqs do
6          if r.lineage has m in PFM then
7              m_seq = r.lineage.get(m)
8              wait(durable_seqs[m] ≤ m_seq)
9      for nm in NFM do
10         send (nm, ⟨notify, my_id, reqs.last.out_seq⟩)
11 upon receiving⟨notify, prev_model, seq⟩
12     set (durable_seqs[prev_model] = seq)
```

lineage information of each client request in the DAG, as shown in Algorithm 1. Specifically, when a successor mode M_{succ} receives a predecessor M_{pred}'s i^{th} output o^i_{pred} as M_{succ}'s j^{th} input r^j_{succ}, it records the mapping between o^i_{pred} and r^j_{succ} (Line 4). This mapping was trivial in a chain as i always equals to j, but the mapping is essential in a general DAG with sequence interleaving (e.g., Figure 1). When M_{succ} releases an output for a processed request, it also carries the request's lineage information with the output (Line 7). In essence, for a request in the service graph that has been processed by a series of models, the request's lineage records the requests sequence number at each model processing it.

Second, in the simplified chain, a stateful model only has one Previous statFul Model (PFM, see §IV-A for definition); in a general DAG, however, a stateful model has multiple PFMs. Therefore, when the backup of this stateful model wants to apply the state generated for a batch of requests, it needs to ensure that, all states generated by these requests in all its PFMs are durable (Line 4-8 in Algorithm 2). After the backup applies this state, it notifies all its next stateful models. Although this mechanism needs a downstream model to wait multiple upstream models' states being durable, it does not introduce overhead in the normal case, as the of upstream models's state propagation are in parallel with downstreams' computation. This waiting is to ensure correctness and is only triggered during network anomalies: an upstream's state propagation is so slow that it cannot finish even after a downstream model finishes both execution and propagation.

To ensure global consistency to a client, HAMS's frontend needs to buffer an output to the client until all stateful models' states generated by this request are durable. Effectively, the frontend can be regarded as a special model, whose state durable action is sending service outputs to clients.

E. Recovery Protocol for General Service Graphs

If an upstream proxy incurs an RPC timeout when trying to send its output to a downstream proxy, the upstream proxy suspects the downstream proxy to have failed and reports the suspected failure to HAMS's manager (§III-A). We first explain HAMS's recovery protocol for single-host failures. If a stateless model (say M_l) is suspected to fail by its predecessors, HAMS takes a hot standby (§V) of the model (M'_l) and reconstructs its dataflow. HAMS first contacts all M_l's successors' proxies to collect the lineage information of requests received from M_l. From this information, HAMS can achieve two important information: the max sequence number (`max_out`) of the original M_l's output, and the max sequence M_l processed for each predecessor model. Then, HAMS sets the output sequence number of the newly launched M'_l to `max_out`, and lets each predecessor model resend outputs to M'_l from their max sequence.

If a stateful model (M_f) is suspected to fail by its predecessors, HAMS first contacts its backup to get the max output sequence (`max_seq`). Then, HAMS checks the primary of all downstream models in parallel to get a list of stateful models that have processed requests whose lineage shows a larger sequence than `max_seq` from M_f (§IV-C). For M_f and models in the list, HAMS promote their backup and lets the predecessors start to resend requests according to the lineage information. Note that HAMS retrieves the full internal state of a stateful model (§IV-B) but not state updates. Therefore, when a backup is promoted as the new primary, the old primary can immediately work as a backup by overwriting its state with the new primary's.

HAMS's recovery protocol for correlated failures (i.e., failures of adjacent models) can be easily deduced by integrating the two recovery protocols for single failures, as the essential step during a failover is to reconstruct the dataflow, for a newly launched stateless model or a new stateful primary, based on the lineage information provided by a failed model's successors. If a contacted successor model fails, this information can be conservatively re-constructed from its backup (for stateful) or its successor (for stateless) models. During the recovery of correlated failures, duplicate intermediate requests may be resent, but HAMS can discard them trivially because intermediate requests have sequence numbers. Overall, HAMS can tolerate arbitrary failures on stateless models, and at the same time, can tolerate one point of failure (primary or backup) for each stateful model.

F. Proof Sketch of Correctness

In this subsection, we prove that HAMS's failover ensures global consistency for a general service DAG. If multiple services share one model, they can be merged as a single service DAG. To prove global consistency, it is sufficient to prove the following statement. Given that a stateful model M_p that processed r^i_p, generated s^i_p, and released o^i_p to downstream models; o^i_p is transformed by a series of stateless models, and is processed as r^j_q on next stateful model M_q. If the resultant state s^j_q is durable, then after a failover, s^i_p, o^i_p will be recovered unchanged. This statement is easily proved in NSPB: s^i_p, o^i_p will be recovered consistently because HAMS's protocol lets a downstream model wait until all its upstream models' states are durable (§IV-C). Therefore, s^j_q being durable derives that s^i_p, o^i_p are durable and can be trivially recovered.

This statement is *sufficient* for global consistency because if s^j_q is not durable yet, the backup of M_q contains a past state s^k_q with $k < j$. Therefore, even if the upstream M_p generates a different (non-deterministic) state and output for processing

r_p^i during failover, HAMS can safely discard s_q^j by promoting the backup of M_q as the new primary.

A concern may be whether r_q^j will be recovered consistently. Although o_p^i stays consistent, r_q^j may diverge due to the non-determinism of intermediate stateless operators. However, the subtle point is that, these intermediate models are stateless, so HAMS does not need to let them redo the computation to generate internal states. As the conditions already state that s_q^j is durable, we do not need to care about r_q^j.

V. IMPLEMENTATION

HAMS's implementation has about 11K LoC, and HAMS's frontend and gRPC servers are based on Tensorflow Serving's code base [77]. In HAMS, all the operators and HAMS's components (i.e., frontend, manager, and proxies) are running in containers. Overall, HAMS's NSPB implementation has two major components. First, the proxy (§III-A) of each operator is written in C/C++ and agnostic to the ML operator served. Communications among proxies are using gRPC [23] and Protobuf [64]. Second, for each operator that deploys an ML model, HAMS provides a library that implemented a gRPC server and API for an asynchronous state retrieving mechanism, described in §IV-B. We implemented HAMS's and all the models based on PyTorch v1.2 and CuDNN v10.0.

For each ML model deployed in HAMS, the ML developer needs to implement two interfaces, `initialize()`, `run()`. The `initialize()` will initialize the model (i.e., load the model from disk to GPU memory) and returns references of tensors that contains a model's state (e.g., `model.parameters.data()`). A developer will implement the `run()` interface that takes the inputs of computation (either training request or inference requests), and an `event` object to explicitly identify the compute and update stage in HAMS. We used PyTorchs API (i.e., `torch.tensor.to_()`) for non-stop copy of tensor memory (the state) between GPU and CPU.

In our evaluation, integrating each of the evaluated ML operators to HAMS added only 4-10 LoC with HAMS's API. To do fast failover for stateless operators, in HAMS, we run one hot standby for each type of stateless operators among all service graphs running in one cluster. For instance, HAMS runs one hot standby with all necessary ML libraries loaded for all KNN operators. By doing so, recovering one stateless operator in a service graph requires only loading the operator's parameters, which takes only hundreds of milliseconds. We used this optimized stateless standby operator setting for all systems we evaluated to make a fair comparison on recovering failed stateless operators (§VI-D).

VI. EVALUATION

A. Evaluation Setup

Our evaluation was done on a GPU farm of five hosts with in total 100 CPU cores and 20 Nvidia RTX2080TI graphic cards. Each graphic card is connected to the host with PCIe 3.0. The `ping` latency across hosts is 0.17ms and the network bandwidth is 40 Gbps. We evaluated HAMS with 25 operators of mature and well-known ML models, and parameters are trained by us using well-known datasets (e.g., CIFAR-10 [38]) on PyTorch. We used these operators to build six practical ML services: sentiment and subject analysis (SA), stock prediction (SP), auto-pilot (AP), image query (IQ), online learning of a VGG19 model (OL(V)) with a heavyweight model size (548.05MB) and online learning of a MobileNet (OL(M)) with a lightweight model size (13.37MB). The online learning service fine-tunes an image labeling model and infers the image context (e.g., a man is with a dog) both for the inference images and the labeled training images. These services are often online and mission-critical, so HAMS's high availability support is desirable for these services.

Figure 8 and Figure 9 describes the semantic of each service. All models' algorithms are all well-known and taken from third-party. All data sets are downloaded from third-party databases or Internet, including Kaggle Speech [32], NYSE Stock data [57], Twitter data [78], autopilot data [17], UTKFace data [79], and CIFAR-10 [38].

We compared HAMS with three systems. To evaluate HAMS's overhead, we implemented a bare metal system by disabling all fault tolerance features of HAMS. To evaluate the effectiveness of HAMS's NSPB protocol, we implemented a HAMS-Remus system. In HAMS-Remus, for each stateful operator, we applied a primary-backup protocol following Remus [13]: after processing each request batch, the primary stops, copies the state update into its memory buffer, holds the output until the state is successfully received by the backup.

Same as HAMS, HAMS-Remus is a white-box approach that replicates only a model's state (i.e., parameters) and safely ignores intermediate computation results or framework memory (e.g., PyTorch). Therefore, we considered our evaluation between HAMS and HAMS-Remus fair. Black-box Remus (Remus-VM) will run slower than HAMS-Remus, because Remus-VM must record all dirty GPU memory pages (§IV-B) and replicate unnecessary memory (e.g., intermediate computation results). In practice, only a portion of dirty GPU memory pages store a stateful models state (parameters) [22].

Since the relevant system Lineage Stash (LS) [84] is not open source, to compare HAMS's performance overhead with LS, we implemented LS on HAMS's code base according to the LS paper: the proxy of each operator logs requests and requests' interleaving locally, and propagates the requests' interleaving along with its output; the local buffer in each proxy is periodically and asynchronously checkpointed to a global storage; each stateful operator is periodically (per 150 requests, the smallest default setting from the LS paper) checkpointed for recovery. When a failure occurs, the stateful operator is replayed from the latest checkpoint with the logged requests and interleaving. We focus on these questions:

- §VI-B : How is HAMS's normal case overhead compared to the relevant systems?
- §VI-C : How effective is NSPB in maintaining HAMS's low performance overhead?
- §VI-D : How is HAMS's recovery time compared to LS and HAMS-Remus?
- §VI-E : What are the limitations of HAMS?

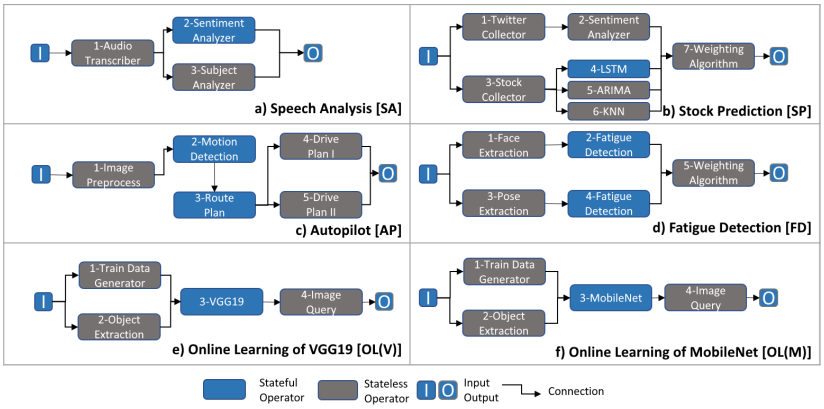


Fig. 8: Service graphs of the six services we evaluated.

	ID	Size(MB)	Input	Output	Model
SA	1	793	Audios	Word list	LSTM [20]
	2	121.7	Words	Sentiment	LSTM [85]
	3	121.7	Words	Subject	LSTM [85]
SP	2	34.8	Words	Sentiment	LSTM [85]
	4	15.3	Stock	Stock	LSTM [53]
	5	N/A	Stock	Stock	ARIMA [9]
	6	N/A	Stock	Stock	KNN [87]
AP	1	90.9	Images	Tagged image	InceptionV3 [75]
	2	375.9	Images	Motion	DeconvLSTM [20]
	3	13.2	Map	Route plan	LSTM [20]
	4	6.2	Meta	Car move	A* search [44]
	5	29.6	Meta	Car move	CNN [8]
FD	1	90.92	Images	Tagged image	InceptionV3 [75]
	2	199.7	Images	Detect result	DeconvLSTM [20]
	3	90.92	Images	Tagged image	InceptionV3 [75]
	4	209.3	Images	Detect result	DeconvLSTM [20]
OL-V	3	548.05	Images	Labeled images	VGG19 [73]
OL-M	3	13.37	Images	Labeled images	MobileNet [28]

Fig. 9: Services' operators.

B. Performance

Figure 10 shows the latency that a request traverses a service graph (normalized to latency of the bare metal) of the services on four systems, including the bare metal, LS, HAMS, and HAMS-Remus. The request batch sizes of all the six services were 64, a typical setting for ML deployments. Overall, compared with the bare metal, HAMS achieved 0.5% to 3.7% latency overhead, which is comparable to the results reported in the LS paper. HAMS achieved such a low latency overhead due to two reasons. First, its NSPB protocol eliminates the stop-and-buffer delay in every stateful operator in a service graph, enabling the operators to process requests in an efficient pipeline as the bare metal. Second, although HAMS does records requests' lineage information to construct the causal dependency of the states across upstream and downstream operators (§IV-D), HAMS's logging time cost for each batch of requests processed by each operator was at most 2.1ms, much smaller than the processing time of an operator on a batch of requests (typically, hundreds of milliseconds).

HAMS-Remus incurred the highest latency overhead (6.0%-97.7%) because each stateful operator needs to stop-and-copy the state to its local memory for every request batch and to hold the output until the updated state is received by the backup. Moreover, for a serving graph that has multiple stateful operators on one path (e.g., AP), the latency overhead of HAMS-Remus was even higher because the request propagation was delayed by multiple times. For SA, HAMS-Remus's latency overhead was small because in the SA service graph, a stateless operator (i.e., audio transcriber) took the most time (1471.23ms), and HAMS-Remus's fault

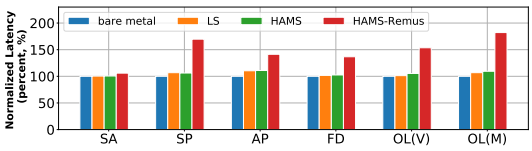


Fig. 10: Normalized latency of six services running on four systems.The request batch size is 64 for all services. The latency is normalized to bare metal in order to fit in one figure. The absolute value of HAMS's latency is in Table I.

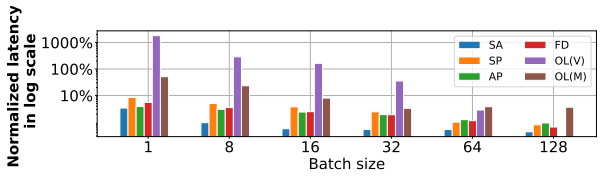
tolerance logic added only 101.23ms to latency.

To further understand HAMS's performance overhead, in Figure 11a, we analyzed the sensitivity of HAMS's latency to the batch sizes of requests on the six services. Overall, when the request batch size of a service increases from 1 to 128, HAMS's latency overhead is greatly reduced. On the batch size of 64 or 128, HAMS's latency overhead was at most 3.8%.

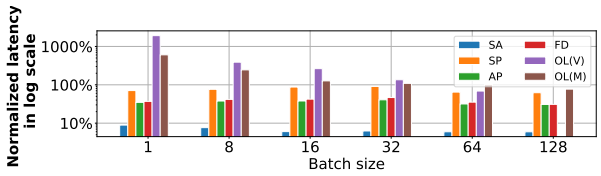
HAMS's latency overhead can be broken down into two parts depending on the nature of a service. The first part is the online learning services (OL(V) and OL(M)). When the batch size was small (e.g., 1), HAMS's latency overhead was almost as high as HAMS-Remus. The reason is that the model operator's state (i.e., model parameters) is static with any request batch size. When the batch size was small, the computation and update stage of the downstream operator (e.g., operator 4 in OL(V)) were both fast (e.g., 12.80ms and 2.43ms in OL(V)), while HAMS's state retrieve time (e.g., 134.52ms in OL(V)) and state delivery time to backup (e.g., 156.43ms in OL(M)) on this operator were the major factor for latency. Therefore, NSPB's non-stop primary-backup was not able to mask the latency overhead on this small batch size. Fortunately, in practice, the batch size of an online learning service is often at least 64 [84], and HAMS's latency overhead was merely 2.8%.

The second part is the inference services (SA, SP, AP, and FD) that contain stateful operators (e.g., LSTM). HAMS's latency overhead was consistently small (less than 10%) when the batch size varied from 1 to 128. The reason is that the size of LSTM's internal state is linear to the request batch size (i.e., each request owns a copy of state). Hence, when the batch size was small (e.g., 1), although the processing time of downstream operators was small (e.g., 17.43ms in FD), and HAMS's state retrieval time (e.g., 11.63ms in FD) and state delivery time to backup (e.g., 2.4ms in FD) of the operator was also small. HAMS's NSPB protocol can mask the latency overhead on all batch sizes. We also evaluated HAMS-Remus in Figure 11b under the same experiment setting. For all combinations of services and batch sizes, HAMS-Remus's latency overhead was in average 5.51X slower than HAMS.

Figure 12 shows the services' throughput in four systems, normalized to bare metal. HAMS incurred little throughput overhead. For SA, HAMS-Remus also had little throughput



(a) HAMS



(b) HAMS-Remus

Fig. 11: Latency overhead of the six serviced deployed with HAMS and HAMS-Remus, with varied request batch size. For batch 128 setting, VGG19 is N/A because a single GPU's memory cannot hold the whole computation.

downgrade because the audio transcriber operator is the throughput bottleneck of SA, and hence the SA's throughput is not affected by HAMS-Remus's fault tolerance logic. Overall, we consider HAMS's normal case performance reasonable for deploying real-world ML services.

In summary, HAMS incurs little normal case performance overhead if the following two conditions are met. First, in a stateful model, the time taken by the computation stage of processing the $(n+1)^{th}$ batch of requests is longer than the time taken by HAMS to retrieve this model's state updated by the n^{th} batch. If so, after processing a batch of requests, NSPB can retrieve the model's state in parallel with the computation stage of the next batch (IV-B).

Second, each stateful model has downstream models in the service DAG, so that the delivery of the stateful model's state can be done in parallel with the processing of downstream models (IV-C). Otherwise, if a stateful model is the last model in the DAG (e.g., in the AP service), its outputs must be buffered at the frontend against a client until the state updated by this client's request is delivered to the model's backup. HAMS incurred little performance overhead in SA because SA's latency is dominated by the first operator. In our evaluation, we found both conditions were often met when the batch size of a service was no less than 64, a typical setting in real-world deployments.

C. *Effectiveness of* HAMS*'s Components*

HAMS's NSPB protocol contains two components for low performance overhead: retrieving a model's state without stopping (§IV-B) and fast output releasing without waiting for state delivery to the backup (§IV-C). To analyze the effectiveness of NSPB's two components, in Table I, we evaluated the six services' latencies with batch size 64 under two settings: (1) disabling the fast output releasing and buffering the output until the state delivered to the backup (HAMS-S1), and (2) disabling the non-stop state retrieving but still enabling the fast output releasing (HAMS-S2).

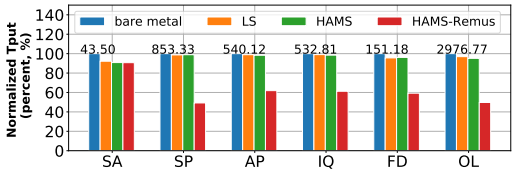


Fig. 12: Throughput of six services, all are normalized to bare metal. Bare metal's absolute value (# requests processed per second) is on the figure. The request batch size is 64.

The results show that HAMS-S1 incurred at most 53.94% latency slowdown compared with HAMS in the normal case, and that HAMS-S2 incurred at most 57.05% latency slowdown compared to HAMS. Nevertheless, both HAMS-S1 and HAMS-S2 had lower latency than HAMS-Remus, which indicated that the both components (§ IV) are essential.

	HAMS	HAMS-S1	HAMS-S2	HAMS-Remus
SA	1604.66ms	1640.32ms	1664.12ms	1671.88ms
SP	123.02ms	152.92ms	172.41ms	210.45ms
AP	289.06ms	320.18ms	349.80ms	375.94ms
FD	224.94ms	252.05ms	271.42ms	300.86ms
OL(V)	292.47ms	450.23ms	426.13ms	508.64ms
OL(M)	22.31ms	32.90ms	35.04ms	43.26ms

TABLE I: Effectiveness of HAMS's components.

D. *Recovery Time*

We compared the recovery time cost of three fault tolerance systems (HAMS, HAMS-Remus, and LS). For HAMS and HAMS-Remus, we randomly picked one stateful operator in each of the six services (under batch size 64 setting) and killed its primary. For LS, we killed the primary operator in each service at the 50^{th} batch of requests from its latest checkpoint (LS does a checkpoint per 150 batches of requests).

Overall, both HAMS and HAMS-Remus achieved a sub-second level of recovery time. HAMS and HAMS-Remus both have a hot standby backup for each stateful operator, so the recovery time was mainly composed of failure discovery, recovery protocol (§IV-E), and backup handover. LS achieved a minute-level recovery time. The recovery time was mainly composed of operator's checkpoint loading (i.e., initialization of an ML model) and the replay time. The replay time depended on the timing of failures, and we chose one-third of LS's checkpoint interval (150 batches of requests). LS's paper [84] also reports minute-level recovery time. We set LS's checkpoint interval to every 1 batch of requests, so that LS can have fast recovery time. LS incurred 81% latency overhead on average, as LS essentially became HAMS-Remus: after a stateful operator processes a request batch, LS needs to stop, copy, and transfer its state to another host.

We also killed a stateless operator in each service and found that HAMS, LS, and HAMS-Remus shows similar recovery time with on average of 320.45 milliseconds, as the recovery time for all three systems was dominated by the time to update a service graph's topology by adding the stateless hot standby operator and by updating the proxies' logic (§V).

To evaluate HAMS's recovery on correlated failures, we did three experiments on the SP service as a complicated case and the AP service for the adjacent stateful models' case (§IV-C). First, we killed O3 (i.e., operator ID#3 in

	HAMS	HAMS-Remus	LS
SA	116.12ms	109.23ms	124.43s
SP	142.43ms	123.12ms	32.04s
AP	150.01ms	119.01ms	56.94s
FD	143.25ms	127.34ms	47.82s
OL(V)	254.19ms	315.42ms	62.10s
OL(M)	134.74ms	141.84ms	21.09s

TABLE II: Recovery time of HAMS components (§IV).

Figure 8) and O4, a stateless model and a stateful model, in the SP service, and we collected an average recovery time of 344.79ms, dominated by the time taken to relaunch a new stateless O3. Second, we killed the primaries of O2 and O3, two adjacent stateful models in AP, and collected an average recovery time of 172.24ms, around 20ms longer than the recovery time of only killing one of them because HAMS iteratively identifies suspected failed models (§IV-E) so it needs an additional timeout (20ms) to suspect the second failures. HAMS's recovery time on these two experiments of correlated failures was just a little larger than that of a single failure as HAMS recovers each failed model in parallel (§IV-E).

Third, we triggered the case of Figure 6 using AP by delaying state delivery of O2 and then killed its primary, and we also killed O3's backup at the same time. In this extreme case, HAMS still ensures global consistency: we checked each operator's proxy log and found both the new primaries of O2&O3, and their downstream models O4, O5, and clients, did not receive conflicting requests. However, this case incurred a recovery time of 731.24ms in average. This time was mainly taken by O3's primary for stopping its current execution on GPU and rolling back to a previous state (§IV-C), which is much larger than the time for promoting a backup to primary (Table II). This result suggests that promoting downstream models' backups is indeed more efficient than rolling back their primaries when an upstream stateful model fails, aligning with NSPB's design choice (§IV-C).

E. Limitations

HAMS has three limitations. First, HAMS needs 2X resource for replicating each stateful model. We deem this resource usage worthwhile because it is essential to ensure sub-second failover time for mission-critical services. In practice, only a small portion of models in a service graph is stateful and needs the extra resource usage. Second, HAMS requires developers to identify the computation and update stages of a stateful model. However, with HAMS's API, this identification is simple and needs only 4-10 lines of code for each evaluated model (§V). Third, HAMS's NSPB protocol is only applicable to ML operators that follows the compute-then-update (i.e., a clear boundary between compute and update) manner (§II-B), and typical ML operators (e.g., neural networks) follow this manner. Studying other ML algorithms is left as future work. HAMS is optimized for ML-context-aware, white-box replication deployed in a service graph manner, while LS and Remus do black-box replication for general services.

VII. RELATED WORK

ML serving systems, including Clipper [11], Michelangelo [49], Tensorflow-Serving [58], TensorRT [81], Pretzel [42], and Ray [50], simplifies deployment of ML models by providing a web server frontend [11], [58], [81], and runtime caching [11], [42], [58] or batching [11], [42], [50], [58]. However, none of them focus on proving high availability, so HAMS can be integrated to all these systems.

GranSLAm [34] and Inferline [10] optimizes deployment of ML service graphs to meet service level latency requirements, but they do not handle failures, so HAMS can benefit these two systems. ParM [37] uses a parity model trained with erasure code [45] to reconstruct prediction results of a failed model to reply the clients timely. ParM does not support stateful models.

State Machine Replication (SMR) [12], [41], [43], [48], [59], [63], [80] models a stateful application as a *deterministic* state machine and feed replicas of the application with the same sequence of requests. HAMS uses SMR to replicate its frontend as it is deterministic. To handle non-determinism, crane [12] and Rex [24] make thread scheduling deterministic [12], [24]. These systems are not suitable for ML models running on GPU as GPU's scheduling is non-deterministic in the architecture level [30]. Plover [83], Eve [35], and Colo [15] compares state update or outputs among replicas, and invoke a state transfer on divergence. These systems fallback to Remus for ML service graph as an ML model usually update all its state after each batch of requests.

Fault tolerance systems for Dataflow [19], [51], [52], including distributed ML training [1], [69], stream processing [2], [6], [86], micro-service systems [2], [3], [21], and network function virtualization [36], [39]. Among these systems, Tensorflow [1], Apache Flink [6] Naiad [51], and Horovod [69] uses periodical global checkpoint at runtime and invokes a coordinated rollback on failures, leading to a long recovery time [84]. Guard [40] further reduces normal case overhead with asynchronous checkpoints. Ray [50], CIEL [52], Drizzle [82], and Noria [21], Manetho [19], and Lineage Stash [84] records the runtime lineage information to reconstruct lost state on failures. These systems do not handle operators' internal non-determinism during replay.

VIII. CONCLUSION

We presented HAMS, an efficient system for deploying highly available ML service graphs. NSPB combines the conceptual strengths of primary-backup and checkpoint-replay to meet all the three fault tolerance requirements (§I). HAMS can serve as a recovery component in existing ML serving systems, greatly improving their reliability. HAMS's source code is released on github.com/hku-systems/hams.

ACKNOWLEDGMENT

Thank all reviewers for their valuable comments. The work is funded by grants partly from the Huawei Innovation Research Program (HIRP) Flagship, HK RGC ECS No.27200916, HK RGC GRF No.17207117, No. 17202318, Croucher Innovation Award, and NNSF China No.61802358.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[3] R. C. Aksoy and M. Kapritsos. Aegean: replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 385–398. ACM, 2019.

[4] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[5] Transposed Convolution Operator source code of Caffe2. https://github.com/pytorch/pytorch/blob/master/caffe2/operators/conv_transpose_op_cudnn.cc#L330.

[6] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[8] L. O. Chua and T. Roska. The cnn paradigm. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 40(3):147–156, 1993.

[9] J. Contreras, R. Espinola, F. J. Nogales, and A. J. Conejo. Arima models to predict next-day electricity prices. *IEEE transactions on power systems*, 18(3):1014–1020, 2003.

[10] D. Crankshaw, G.-E. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov. Inferline: Ml inference pipeline composition framework. *arXiv preprint arXiv:1812.01776*, 2018.

[11] 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)*, pages 613–627, 2017.

[12] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[14] Y. Dong, M. Xue, X. Zheng, J. Wang, Z. Qi, and H. Guan. Boosting gpu virtualization performance with hybrid shadow page tables. In *2015 USENIX Annual Technical Conference USENIX ATC 15)*, pages 517–528, 2015.

[15] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[16] M. Dowty and J. Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[17] Auto Driving Dataset. https://github.com/SullyChen/driving-datasets/blob/master/README.md.

[18] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[19] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, (5):526–531, 1992.

[20] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. 1999.

[21] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 213–231, 2018.

[22] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[23] gRPC. https://grpc.io.

[24] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[25] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[26] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[27] Horovod. https://github.com/horovod/horovod.

[28] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[29] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[30] H. Jooybar, W. W. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt. Gpudet: a deterministic gpu architecture. *ACM SIGARCH Computer Architecture News*, 41(1):1–12, 2013.

[31] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[32] Kaggle Speech Accent Dataset. https://www.kaggle.com/rtatman/speech-accent-archive/.

[33] Kaggle Vehicle Number Plate Detection. https://www.kaggle.com/datature/vehicle-number-plate-detection.

[34] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 34. ACM, 2019.

[35] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[36] J. Khalid and A. Akella. Correctness and performance for stateful chained network functions, 2018.

[37] J. Kosaian, K. Rashmi, and S. Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 30–46. ACM, 2019.

[38] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[39] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu. Reinforce: Achieving efficient failure resiliency for network function virtualization based services. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 41–53. ACM, 2018.

[40] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment*, 1(1):574–585, 2008.

[41] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[42] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626, 2018.

[43] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 467–483, 2016.

[44] J. Li and X.-x. Sun. A route planning's method for unmanned aerial vehicles based on improved a-star algorithm [j]. *Acta Armamentarii*, 7:788–792, 2008.

[45] W. Lin, D. M. Chiu, and Y. Lee. Erasure code replication revisited. In *Proceedings. Fourth International Conference on Peer-to-Peer Computing, 2004. Proceedings.*, pages 90–97. IEEE, 2004.

[46] M. Lu and T.-c. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 534–543. Citeseer, 2009.

[47] A. I. Maqueda, A. Loquercio, G. Gallego, N. García, and D. Scaramuzza. Event-based vision meets deep learning on steering prediction for self-driving cars. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5419–5427, 2018.

[48] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[49] Uber Michelangelo. https://eng.uber.com/michelangelo/.

[50] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[51] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[52] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2011.

[53] D. M. Nelson, A. C. Pereira, and R. A. de Oliveira. Stock market's price movement prediction with lstm neural networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1419–1426. IEEE, 2017.

[54] Nvidia CuDNN User Guide (Reproducibility). https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#reproducibility.

[55] Determinism in Deep Learning (S9911). https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9911-determinism-in-deep-learning.pdf.

[56] Nvidia tensorflow-determinisitc project. https://github.com/NVIDIA/tensorflow-determinism.

[57] NYSE Stock. https://www.nyse.com/listings_directory/stock.

[58] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[59] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[60] S. Pandi, F. H. Fitzek, C. Lehmann, D. Nophut, D. Kiss, V. Kovacs, A. Nagy, G. Csorvasi, M. Tóth, T. Rajacsis, et al. Joint design of communication and control for connected cars in 5g communication systems. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–7. IEEE, 2016.

[61] A. Paszke, S. Gross, S. Chintala, and G. Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, 6, 2017.

[62] PCI Express. https://en.wikipedia.org/wiki/PCI_Express.

[63] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[64] Protobuf. https://developers.google.com/protocol-buffers/.

[65] Convolution Operator source code of PyTorch. https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/cudnn/Conv.cpp#L647.

[66] Transposed Convolution Operator source code of PyTorch. https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/cudnn/Conv.cpp#L1071.

[67] PyTorch Reproducibility. https://pytorch.org/docs/stable/notes/randomness.html.

[68] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[69] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[70] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2004.

[71] S. Shalev-Shwartz et al. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194, 2012.

[72] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 227–240. ACM, 2015.

[73] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[74] Stock Market Price Predictor using Supervised Learning. Stock Market Price Predictor using Supervised Learning.

[75] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[76] Convolution Operator source code of Tensorflow. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/stream_executor/cuda/cuda_dnn.cc#L250.

[77] tensorflow/serving github repo. https://github.com/tensorflow/serving.

[78] Twitter. https://twitter.com/.

[79] UTKFace Dataset. https://susanqq.github.io/UTKFace/.

[80] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[81] H. Vanholder. Efficient inference with tensorrt, 2016.

[82] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[83] C. Wang, X. Chen, W. Jia, B. Li, H. Qiu, S. Zhao, and H. Cui. Plover: Fast, multi-core scalable virtual machine fault-tolerance. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI'18)*. USENIX Association, 2018.

[84] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 338–352. ACM, 2019.

[85] Y. Wang, M. Huang, L. Zhao, et al. Attention-based lstm for aspect-level sentiment classification. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 606–615, 2016.

[86] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

[87] H. Zhang, A. C. Berg, M. Maire, and J. Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2126–2136. IEEE, 2006.