Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving

Luping Wang*
HKUST, Alibaba Group
lwangbm@cse.ust.hk

Lingyun Yang*
HKUST, Alibaba Group
lyangbk@cse.ust.hk

Yinghao Yu
Alibaba Group, HKUST
yinghao.yyh@alibaba-inc.com

Wei Wang
HKUST
weiwa@cse.ust.hk

Bo Li
HKUST
bli@cse.ust.hk

Xianchao Sun
Alibaba Group
xianchao.sxc@alibaba-inc.com

Jian He
Alibaba Group
jian.h@alibaba-inc.com

Liping Zhang
Alibaba Group
liping.z@alibaba-inc.com

Abstract

Machine learning models are widely deployed in production cloud to provide online inference services. Efficiently deploying inference services requires careful tuning of hardware and runtime configurations (e.g., GPU type, GPU memory, batch size), which can significantly improve the model serving performance and reduce cost. However, existing auto-configuration approaches for general workloads, such as Bayesian optimization and white-box prediction, are inefficient in navigating the high-dimensional configuration space of model serving, incurring high sampling cost.

In this paper, we present Morphling, a fast, near-optimal auto-configuration framework for cloud-native model serving. Morphling employs model-agnostic meta-learning to navigate the large configuration space. It trains a *meta-model* offline to capture the general performance trend under varying configurations. Morphling quickly adapts the meta-model to a new inference service by sampling a small number of configurations and uses it to find the optimal one. We have implemented Morphling as an auto-configuration service in Kubernetes, and evaluate its performance with popular CV and NLP models, as well as the production inference services in Alibaba. Compared with existing approaches, Morphling reduces the median search cost by 3×-22×, quickly converging to the optimal configuration by sampling only 30 candidates in a large search space consisting of 720 options.

*The first two authors made equal contributions to this work, which was done while they were doing research internship in Alibaba.

SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486987

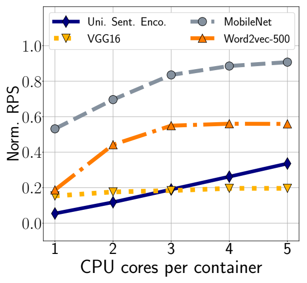
CCS Concepts: • **Information systems → Enterprise resource planning**.

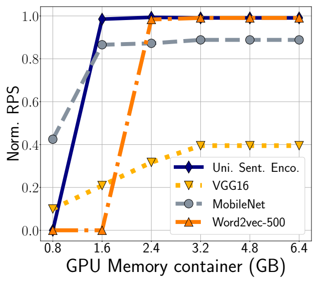
Keywords: Cloud Computing, Model Serving, Meta-Learning, Auto-Configuration

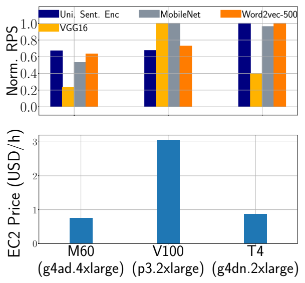
1 Introduction

Tech companies are increasingly building large Machine-Learning-as-a-Service (MLaaS) cloud for *model training* and *inference serving*. In a typical MLaaS workflow, developers design and train ML models offline with large datasets; the trained models are then published in the cloud to provide online inference services, typically running in containers that can be queried by users to make predictions for given inputs [21, 28, 29, 43, 50, 70]. As MLaaS cloud serves massive volumes of inference requests (e.g., tens of trillions per day in Facebook [30]), the majority of resources and costs are dedicated to inference serving (e.g., up to 90% in AWS [2]).

However, efficiently deploying inference services in the cloud is challenging. Given a trained model, cloud operators need to specify *hardware configurations* for each model serving container, such as CPU cores, GPU type, GPU memory, and GPU share (if GPU sharing is supported), as well as *runtime configurations* such as batch size. Together, they form a large, high-dimensional configuration space. The choice of configurations largely determines the model serving performance and cost. In our testbed experiments, we find that a good configuration yields over 10× request throughput than a bad one (see Figs. 1 and 2). We also observe, in the production cloud of Alibaba, that inefficient model serving configurations result in low resource utilization, with nearly 80% of CPUs and GPU memory allocated but not used.


(a) Impact of CPU cores


(b) Impact of GPU memory


(c) Impact of GPU types

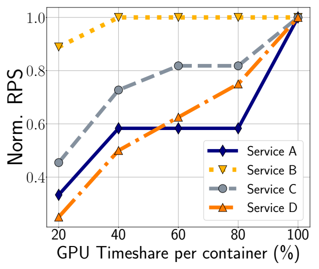

(d) Impact of GPU timeshare

Figure 1. Hardware configurations largely determine the service throughput, measured by requests per second, or RPS. For each inference service, the throughput is normalized by the highest RPS. Models in (a)-(c) are provided by TensorFlow model zoo [8, 15], running on EC2; models in (d) are proprietary, running in a production cloud supporting GPU time-sharing.

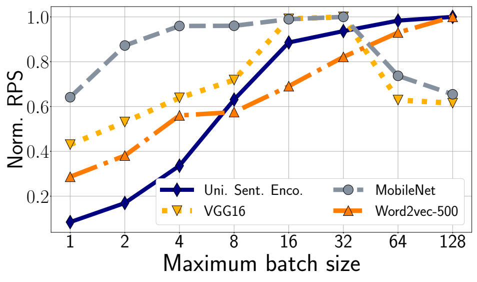


Figure 2. Request batch size versus service throughput. The throughput of each service is normalized by the highest RPS.

In light of these problems, our goal is to design an efficient approach that can quickly find the optimal configuration of an inference service in a large search space. Despite a rich body of work on model serving and workload auto-configuration, achieving this goal remains elusive. Existing model serving systems [10, 16, 21, 28, 29, 50, 62, 70, 73] are designed to streamline model deployment with enriched features of request scheduling, auto-scaling, and auto-selection of model variants. These systems do not tune resource configurations of model serving containers, but simply follow the specifications given by developers. Prevalent auto-configuration techniques, such as Bayesian optimization [18], linear regression [36, 61], and transfer learning [27], perform well in tuning general cloud workloads in a *low-dimensional* configuration space (e.g., determining the number/type of VMs [18, 35]). Yet, these methods become inefficient as the search space increases [33, 49, 65], incurring a large sampling overhead to find the optimum (see §6).

In this paper, we present Morphling, a fast, near-optimal auto-configuration framework for cloud-native model serving. Our key observation is that hardware and runtime configurations have a *general performance impact* to a wide variety of inference services running different ML models. For example, to run an inference service, there is a minimum requirement of GPU memory to fully load the serving model (see Fig. 1b); further increasing GPU memory allows it to serve larger request batches with higher throughput; yet, such improvement diminishes as the bottleneck shifts from GPU memory to other resources like CPUs. The general performance impact of configurations leads to resembling configuration-throughput planes of different models (Figs. 3 and 4): despite the varying turning points and scales, the shapes of these planes show a similar tendency.

Based on this observation, we formulate optimal configuration search as a *few-shot learning* problem [44, 60, 63] and solve it with the recently developed *model-agnostic meta-learning* (MAML) technique [25]. In particular, Morphling trains a meta-model offline that captures how the inference serving performance may change generally under varying hardware and runtime configurations (see Figs. 3d and 4d). The meta-model provides an informative prior to configuration search, and is used as a good initialization of the learning process. Given a new inference service, Morphling performs online few-shot learning: it samples a small number of configurations and uses the profiled results to adapt the meta-model to the new service. The adapted meta-model can be used to accurately predict the service performance, enabling a fast search for the optimal configuration.

We have implemented Morphling as an easy-to-use auto-configuration service in Kubernetes [6] with around 5,000 lines of Golang code. Morphling exposes common interfaces that abstract away the heterogeneity of model serving frameworks and service deployments. Users implement the interfaces by specifying the serving model, the tunable configuration parameters, the optimization objectives, and the sampling budget; Morphling then automatically tunes configurations to attain the optimal serving performance, within the specified sampling budget.

We evaluate Morphling on Amazon EC2 with 42 models provided by TensorFlow model zoo [8, 15] for image classification and language processing. Morphling quickly identifies the optimal configuration by sampling less than 5% of a large search space consisting of 720 options. In comparison, existing auto-tuning approaches require to sample 3×-22×

more configurations before find the optimal ones. We also evaluate Morphling with 30 real-world production inference services in an Alibaba's cluster. Morphling finds the optimal configuration for all services by sampling no more than 19 options out of 100 choices, while existing approaches require at least 60 samplings for guaranteed optimality. We plan to deploy Morphling as the default auto-configuration service in Alibaba's production clusters for efficient model serving at scale.

2 Background

ML Model Serving. Production clouds run a large number of machine learning models to provide online inference service for various AI applications, such as image classification [31, 55, 72], video processing [40, 48], language modeling [42, 57], and recommendation [17, 22]. To deploy an inference service, model developers encapsulate the trained models, the model serving frameworks, and the pre/post-processing pipelines into Docker containers [3]. These containers run for a long time to provide unfailing services, orchestrated by systems like Kubernetes [6]. Users can then query the serving containers through HTTP/RPC APIs to make predictions.

Cloud-Native Model Serving Systems. Many model serving systems have been developed to streamline model deployment in the cloud for improved performance and reduced costs [21, 28, 29, 43, 50, 70]. For example, systems like Clipper [21], INFaaS [50] and Rafiki [62] abstract away the heterogeneity of existing model execution frameworks with a unifying model abstraction. They also support diverse and customizable model deployment strategies. Existing model serving systems also support dynamic batching [28, 70], inference buffering [21, 50], replica auto-scaling [28, 70], and auto-selection of model variants [21, 50]. The recently proposed white-box model serving systems [29, 43] enable model-specific optimizations with model layer sharing and fine-grained GPU scheduling.

Container-Level Configuration Optimization. Optimizing the *container-level* configurations can significantly improve the performance of inference services and reduce their resource provisioning costs. As we will show in §3, a good container configuration with optimized resource allocations (e.g., CPU cores, GPU memory, GPU share, GPU type) and runtime parameters (e.g., batch size) yields over 10× higher inference throughput than a bad one. However, auto-tuning container configurations has received less attention. Systems like INFaaS [50] and Clipper [21] mainly concern the auto-selection of a number of model variants with different architectures implemented in different frameworks. These model variants are often given by developers and deployed in multiple containers, among which the system adaptively chooses one to serve an inference request. This approach cannot be used to find the optimal container configuration.

3 The Need for Configuration Tuning

In this section, we show empirically that the performance of inference services largely depends on the resource and runtime configurations of the serving containers, which require careful tuning. We hence formulate a configuration optimization problem, and discuss the inefficiency of existing approaches.

3.1 Identifying Important Configurations

Inference services usually run in containers. In Alibaba cloud, we measure a container's serving capability with the peak throughput, defined as the maximum requests per second (RPS) it can serve without violating the response-time service-level objective (SLO). The peak RPS can be easily measured using stress-testing tools [4, 5, 14]. Our production system uses the peak RPS to determine the required number of container instances for each inference service such that the overall serving capability is sufficient to accommodate the request demands.

Given a model serving container, cloud operators need to specify its resource and runtime configurations. To quantify how each configuration may affect the serving capability, we profile four open-source ML models in Amazon EC2 [7] and four production inference services. We stress-test their peak RPS (request latency ≤ 1 second) under various configurations. The detailed experimental setup is given in §6.

Resource configuration. Our characterization starts with resource configurations, including CPU cores, GPU memory, GPU timeshare, and GPU type. In our experiments, we change one configuration while keeping the others fixed. Fig. 1 shows the measurement results.

(a) *CPU Cores.* As shown in Fig. 1a, adding more CPU cores to a serving container enables a higher degree of parallelism for data processing and I/O, thus higher RPS. For most inference services, the RPS improvement diminishes with the increase of CPU cores. The only exception goes to Universal Sentence Encoder [20], a popular language processing model, where RPS improves almost linearly and is up to 3×, the largest among all services.

(b) *GPU Memory.* An inference service occupies GPU memory in two ways: static memory occupation for hosting the model, and dynamic memory usage for caching intermediate results. As shown in Fig. 1b, the memory allocation must be large enough to load the model; additional allocations on top of that allows the service to handle larger request batches thus higher throughput [21, 70]. Taking the `Word2vec-500` language model [46] as an example, it requires GPU memory larger than the model size (1.9 GB) to serve requests.

(c) *GPU Timeshare.* Many GPU sharing techniques have been proposed recently to time-multiplex GPUs [38, 41, 64, 69] between multiple ML workloads. In Alibaba, we employ a fine-grained GPU time-sharing approach to isolate the uses of streaming multiprocessors (SMs) between contending containers. In particular, we use CUDA APIs [52] to control each container's GPU timeshare at a fine granularity. As such feature is only available in our production clusters, we conduct experiments with production inference services. Fig. 1d depicts how RPS changes with the allocated GPU timeshare, where we observe up to 10× RPS variation for compute-intensive inference services.

(d) *GPU Type.* The performance-price trade-off between various GPU types further complicates configuration tuning. In Fig. 1c, we compare three commonly used Nvidia GPUs in Amazon EC2 [28]. In particular, `V100`, with steadily-high frequency [39], provides arguably the best performance for image classification models (similar results also reported by MLPerf Inference benchmark [9, 45]). `V100`'s high performance, however, comes with a high cost. In comparison, `T4`'s low price tag and inference-specific optimizations make it usually a better choice for model serving. On the other hand, `M60`, designed for graphics-intensive applications [11], shows no benefit on either performance or cost.

Runtime Parameters. In addition to resources, runtime parameters are also important factors that determine the inference RPS. In this work, we mainly focus on tuning the *batch size* of inference requests. We will provide more discussions on the impact of the other parameters in §7.

(e) *Batch Size.* Batching inference requests is an effective approach to increasing throughput as it can better utilize the parallel computing power of GPUs [21, 50] and amortize the cost of RPC calls and I/O overheads (e.g., copying data to GPU memory). Frameworks like TensorFlow [16] often enforce a fixed maximum batch size to maintain a consistent data layout. Configuring a large batch size is not always beneficial. In Fig. 2, image processing models like `MobileNet` [34] and `VGG16` [55] see performance degradation with a large batch size, as the large input cannot fit into the GPU memory.

3.2 Problem Formulation and Objective

Given a well-trained ML model, the cloud operator needs to find an optimal resource and runtime configuration for performant and resource-efficient model serving. Formally, consider a configuration vector with M tuneable hyper-parameters $\boldsymbol{x} = \{x_1, x_2, \ldots, x_M\}$, where each x_i has a *discrete* search space consisting of n_i candidates. Our goal is to find the optimal configuration $\boldsymbol{x}^*$ that maximizes the objective function $f(\cdot)$ defined by the operator, i.e.,

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x}\in\mathcal{A}} \quad f(\boldsymbol{x}), \tag{1}$$

where $\mathcal{A}$ is the configuration space, which is combinatorial by nature.

For cloud operators, maximizing the container peak RPS and minimizing the resource costs are the two common objectives, which are often at odds. We leave the navigation of such performance-cost trade-off to the operator by allowing it to define its own objective function $f(\boldsymbol{x})$ through system-provided APIs (see §5), e.g., maximizing the service RPS per monetary cost.

3.3 Prior Arts and Their Inefficiency

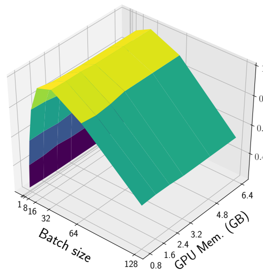
Many auto-configuration approaches have been proposed to tune general cloud workloads. However, none of them can efficiently solve our problem.

Auto-Configuration Using Historical Data. One common approach is to find the optimal configuration based on the workload's past executions [24, 51]. Notably, Google's Autopilot [51] learns the optimal resource allocations for containerized services by analyzing their trace data. Though simple, this approach falls short in performance when applied to model serving. First, ML frameworks like TensorFlow [16] occupy all the available GPU memory when running an ML workload. As a result, monitoring tools like NVIDIA-SMI [12] always observe the full memory usage, even though only a small part is used in inference. Second, historical data only sample *a few* configuration combinations that are already deployed, failing to explore the large search space for optimality. Finally, for newly published or updated services, historical data is not always available, especially for newly published or updated services.

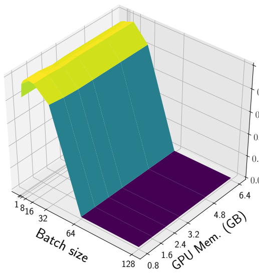
Auto-Configuration Using Search. Another approach to auto-configuration is to sample a small number of configurations for performance evaluation, following some search algorithms. Existing works along this line can be divided into three categories.

1) Black-box search employs sequential model-based optimization (SMBO) [26, 56] to search for the best configuration. During the search process, SMBO builds a regression model (e.g., Gaussian Process) and uses it to fit the configuration-performance curve. The algorithm iteratively samples the next configuration for testing, until the sampling budget runs out. A popular SMBO algorithm is Bayesian Optimization (BO) [26, 56], which has been widely used to tune configurations of cloud workloads (e.g., CherryPick [18], Arrow [35], and Rafiki [62]). While efficient in low-dimensional searching, BO becomes extremely expensive to navigate a large, high-dimensional configuration space [33, 49, 65]. Also, its performance critically depends on the choice of initial sampling: a poor seeding strategy often leads to a sub-optimal result. We will show in §6 that BO is inefficient in tuning inference services, incurring high sampling overhead.

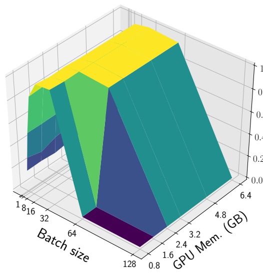
2) White-box prediction is another auto-tuning approach that predicts the performance under a certain configuration and uses it to drive the search process. The key to prediction is to build a regression model with a few samplings using the prior knowledge of how the performance may change with configurations (e.g., linear curve) [61]. However, the high-dimensional configuration-performance plane in our



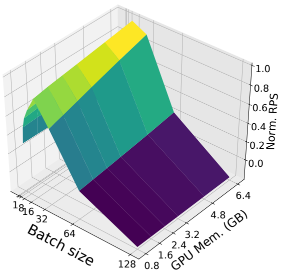
(a) `ResNet101` (image class.)



(b) `ALBERT` (language proc.)

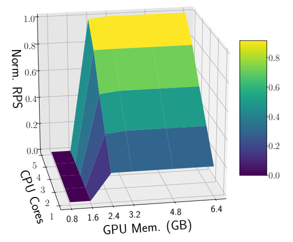


(c) `VGG16` (image class.)

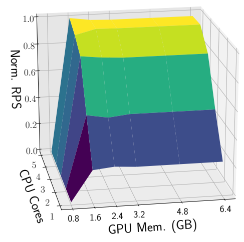


(d) Meta model θ^m.

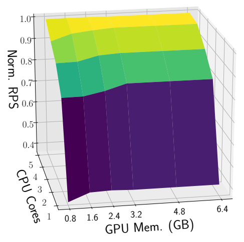
Figure 3. Normalized RPS under different configurations of batch size and GPU memory.



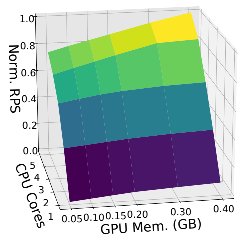
(a) `Uni.Sen.Enc.` (language proc.)



(b) `Word2vec-250` (language proc.)



(c) `ResNet50` (image class.)



(d) Meta model θ^m.

Figure 4. Normalized RPS under different configurations of CPU cores and GPU memory.

problem is so complicated that can hardly be fitted with a few samplings.

3) Similarity-based search measures the similarity between the tuning and benchmarking workloads and uses it to guide the search process. For example, Google Vizier [27] leverages the Gaussian process regressor built from previously studied benchmarks and applies transfer learning to configure a new workload with similar resource usage patterns. Scout [36] and PARIS [67] also configure a new workload by comparing its pre-defined features with benchmarks. These works mainly focus on the *one-to-one* similarity between the current workload and a previously studied benchmark, yet missing the inherent and common performance trends shared by multiple workloads. As we shall show in §6, similarity-based search requires sampling a large configuration space to find the optimal configuration.

4 Algorithm Design

In this section, we first explore the common performance trend for model serving in the previous experiments (§4.1). Utilizing such trend, we present an intelligent configuration tuning algorithm for cloud-native model serving using the meta-learning approach. We explain (1) how it captures the internal features of inference configurations (§4.2), and (2) how the meta-model is used to direct configuration search (§4.3).

4.1 Common Performance Trend

We observe a common trend in Fig. 1 that the service RPS improves by configuring more resources of a type, yet such performance improvement diminishes as the bottleneck shifts to another resource(s) (e.g., from GPU memory to CPU cores). In fact, multiple resources and runtime parameters have *collective impacts* to the service RPS, forming a high-dimensional configuration-RPS plane. Fig. 3 visualizes how the batch size and GPU memory collectively affect the RPS of three inference services using open-source models. For small models like `ResNet101` [31] and `ALBERT` [42], GPU memory has almost no performance impact, regardless of the choice of batch size. For a larger model like `VGG16` [55] (500 MB), the service requires more than 1.6 GB GPU memory to handle a large batch of size greater than 32 (Fig. 3c). Across all three models, enlarging the batch size initially improves the performance, followed by a degradation beyond a turning point. Similar to Fig. 3, Fig. 4 depicts the RPS changes with respect to CPU cores and GPU memory. For a large model like Universal Sentence Encoder [20], a large GPU memory allocation (≥1.6 GB) is needed. Allocating more CPU cores leads to higher RPS, with linear or sub-linear improvement depending on the models.

To summarize, resource and runtime configurations have general performance impacts to a variety of inference services running different ML models (e.g., large GPU memory is needed to load a large model and/or serve a large request batch; adding more CPU cores (marginally) improves the performance), leading to *resembling* configuration-RPS planes—except that the actual turning points and scales may vary from one model to another. This motivates a fast auto-configuration approach using meta-learning techniques. In particular, we train a meta-model offline that captures the common configuration-performance trend for general inference services (illustrated in Figs. 3d and 4d). To tune configurations for a new service, we adapt the meta-model with few-shot learning and use it to guide the tuning process.

4.2 Meta-Model Training

Few-shot Regression. We start by formulating a few-shot learning problem, where we fit the configuration-performance plane with a regression model. Formally, consider a regression task T_i with a mapping function $f_{\theta_i}(\boldsymbol{x})$ that predicts the service performance for an input configuration $\boldsymbol{x}$, where θ_i is the parameterized model tuned for task T_i. The loss function $\mathcal{L}_{T_i}$ is defined as the mean squared error (MSE) between the predicted performance and the real performance y, i.e.,

$$\mathcal{L}_{T_i}(f_{\theta_i}) = \textstyle\sum_{\boldsymbol{x}^j, y^j \sim T_i} ||f_{\theta_i}(\boldsymbol{x}^j) - y^j||_2^2. \tag{2}$$

In a K-shot regression, model θ_i is trained with K sampled input-output pairs $\mathcal{D} = \{\boldsymbol{x}^j, y^j | j = 1, 2, \ldots, K\}$; the objective is to minimize the loss $\mathcal{L}_{T_i}(f_{\theta_i})$. In practice, K is usually set as a small number to minimize the training overhead (e.g., 5% of the search space). Despite such limitation, we still want to achieve accurate prediction with a K-shot regression model.

Model-Agnostic Meta-Learning (MAML). The recently developed Model-Agnostic Meta-Learning (MAML) technique [25] offers a promising solution for K-shot regression. It assumes a set of regression tasks with broadly suitable features, and performs regression model training in two stages. In the first stage, a *meta-model* is trained across a set of regression tasks; the objective is to obtain a meta-model that can quickly adapt to a new, unseen task. In the second stage, a new task is given and the meta-model is adapted to it, ideally converging to a fine-tuned model with a small number of data points. Formally, consider a set of regression tasks $\mathcal{T} = \{T_1, T_2, \ldots T_N\}$ that share common input-output mapping features. Let θ^m be the meta-model trained in the first stage. Given a new task T_i, MAML adapts θ^m to a fine-tuned model θ_i by iteratively updating it with K newly sampled data points $\mathcal{D}_{||K||} = \{\boldsymbol{x}^j, y^j | j = 1, 2, \ldots, K\}$, using stochastic gradient descent (SGD) [19], i.e.,

$$\theta_i = \theta^m - \alpha \nabla_{\theta^m} \mathcal{L}_{T_i}(f_{\theta^m}), \tag{3}$$

where α is the learning rate. We next describe the two training stages in detail.

Stage-1: Meta-Model Training. From the perspective of feature learning, meta-model training essentially builds an internal representation that is broadly applicable to many related tasks. As the meta-model will later be adapted to a new regression task using SGD, it should be trained such that the

Algorithm 1: MAML for Few-Shot Regression

```
Input  : Regression task set of $N$ inference models: $\mathcal{T}$,
         learning rates $\alpha$ and $\beta$, sampling budget $K$, meta
         training episodes $E$
Output : A well-trained meta model with parameters $\theta^m$
1 Randomly initialize $\theta^m$
2 for episode = 1, 2, ..., $E$ do
3     for all $T_i \in \mathcal{T}$ do
4         Sample $K$ datapoints $\mathcal{D}_{||K||} = \{\boldsymbol{x}^j, y^j\}$ from $T_i$
5         Evaluate $\nabla_{\theta^m} \mathcal{L}_{T_i}(f_{\theta^m})$ using $\mathcal{D}$ and $\mathcal{L}_{T_i}$ in Eq. (2)
6         Compute adapted model $\theta_i$ with SGD using Eq. (3)
7     Update meta model $\theta^m$:
        $\theta_m \leftarrow \theta_m - \beta \nabla_{\theta_m} \sum_{T_i \sim \mathcal{T}} \mathcal{L}_{T_i}(f_{\theta_i})$
```

later SGD process can make a rapid progress without overfitting. Therefore, the meta-model θ^m is trained to optimize f_{θ_i} over tasks sampled from $\mathcal{T}$, i.e.,

$$\theta^m \leftarrow \theta^m - \beta \nabla_{\theta^m} \textstyle\sum_{T_i \sim \mathcal{T}} \mathcal{L}_{T_i}(f_{\theta_i}), \tag{4}$$

where β is the learning step in the meta-training stage, and θ_i is the fine-tuned model for task T_i, computed in the second stage following Eq. (3). Algorithm 1 details the training process of the meta-model.

Stage-2: Fast Adaptation. Once the meta-model θ^m is trained, it is used as the initial regression model for a new task T_i, followed by a fine-tuning process to better fit it to T_i (see Eq. (3)). Such fine-tuning converges quickly with only a few data points, as meta-training is meant to enable fast adaptation: it aims to find meta-model θ^m that is *sensitive* to changes in the task, such that a small change of parameters will produce a large improvement on the loss for any T_i. We next develop a novel SMBO (Sequential Model-Based Optimization) approach that directs the search for the optimal configuration using the trained meta-model, along with its fast adaptation.

4.3 Directing SMBO Search with Meta-Model

SMBO is a common approach to configuration tuning [26, 56]. In its standard form, SMBO starts by randomly initializing a regression model, and iterates between fitting the model and using it to determine which configurations to explore next (exploration or sampling). The search stops when the sampling budget runs out. During the search, it is important to strike a balance between *exploration* and *exploitation.* An *acquisition function* is hence defined to navigate such tradeoff, usually by combining both the mean and variance of the predictions made by the regression model [47, 49, 65].

Meta-Model as an Initial Regression Model. Unlike the standard SMBO approaches, we use the trained meta-model θ^m as the initial regression model and adapt it to a new inference service (modeled as a regression task T_i) during the search. Formally, let K be the sampling budget, which is

Algorithm 2: SMBO with Meta Model

```
Input  : A new regression task $T_i$, learning rates $\alpha$,
         sampling budget $K$, meta model $\theta^m$
Output : The optimal configuration $\boldsymbol{x}^*$
1 Initialize $\theta_i \leftarrow \theta^m$, and newly-sampled data set $\mathcal{D} \leftarrow \{\}$
2 for $k = 1, 2, \ldots, K$ do
3     Update regression model $\theta_i' = \theta_i - \alpha \nabla_{\theta_i} \mathcal{L}_{T_i}(f_{\theta_i})$
4     for all $\boldsymbol{x} \in \mathcal{A}$ do
5         Calculate the $f_{\theta_i}(\boldsymbol{x})$ and $\mathrm{Acq}(f_{\theta_i}(\boldsymbol{x}))$ using Eq. (6)
6     $\boldsymbol{x}^k \leftarrow \arg\max_{\boldsymbol{x} \in \mathcal{A}, \boldsymbol{x} \notin \mathcal{D}} \mathrm{Acq}(f_{\theta_i}(\boldsymbol{x}))$
7     Estimate $y^k$ for $\boldsymbol{x}^k$    ▷ Real-world evaluation
8     $\mathcal{D} \leftarrow \mathcal{D} \cup (\boldsymbol{x}^k, y^k)$, $\theta_i \leftarrow \theta_i'$
9 $\boldsymbol{x}^* \leftarrow \arg\max_{\boldsymbol{x}^k \in \mathcal{D}} y^k$
```

the maximum number of configurations that the algorithm can explore. Let θ_i be the refined model for T_i. Given sampling budget K, our algorithm sequentially explores the next configuration by predicting the performance $f_{\theta_i}(\boldsymbol{x}^j)$ for all candidate configurations $\boldsymbol{x}^j$ in the search space.

Exploration-Exploitation Trade-off. Supposing the algorithm performs no exploration but only exploitation, it will always sample the configuration with the highest prediction. This can easily trap the search into a local optimum. We therefore need to strike a balance between exploration and exploitation, which requires the knowledge of prediction confidence. However, a fine-tuned regression model usually has no clue about the uncertainty of the predictions it makes for a configuration, unless the algorithm uses Bayesian posterior covariance to measure the prediction confidence [26, 65].

We solve this problem by defining the prediction confidence with respect to the fine-tuning process. In particular, given a configuration $\boldsymbol{x}$, let $f_{\theta_i}(\boldsymbol{x})$ be the performance prediction made by model θ_i. Upon sampling, the model is updated to $\theta_i' = \theta_i - \alpha \nabla_{\theta_i} \mathcal{L}_{T_i}(f_{\theta_i})$, and the new prediction becomes $f_{\theta_i'}(\boldsymbol{x})$. We define the confidence of $f_{\theta_i'}(\boldsymbol{x})$ as

$$\mathrm{Conf}(f_{\theta_i'}(\boldsymbol{x})) = |f_{\theta_i}(\boldsymbol{x}) - f_{\theta_i'}(\boldsymbol{x})|. \quad (5)$$

That is, a lower Conf(.) indicates a higher confidence. Intuitively, if two sequential regression models θ_i and θ_i' make similar predictions for the same configuration, then we have a high confidence about the results, and vice versa. Following this intuition, we define the acquisition function as an upper confidence bound:

$$\mathrm{Acq}(f_{\theta_i}(\boldsymbol{x})) = f_{\theta_i}(\boldsymbol{x}) + \delta \mathrm{Conf}(f_{\theta_i}(\boldsymbol{x})), \quad (6)$$

where δ is a pre-defined weight knob which is usually a small constant. Algorithm 2 details the search process with the meta-model.

4.4 Why Do We Use Meta-Learning?

In meta-learning, the offline trained meta-model automatically captures the common features and general performance trends of inference services. The meta-model provides an informative and non-overfitting prior for configuration search, and can be adapted online in a few shots to accurately fit a new inference service. The meta-learning approach hence combines the benefits of both black-box search and white-box predictions. It generally applies to a range of inference services and various optimization objectives, while achieving accurate predictions with fine-tuned models automatically learned from the general meta-model. We will show in §6 that the meta-learning approach substantially reduces the required configuration samplings compared to the existing auto-tuning algorithms.

```
type Experiment struct {
     ModelDockerImageID          string
     Objective              ObjectiveSpec
     TunableParameters        []Parameter
     SamplingAlgorithm      AlgorithmSpec
     SamplingBudget              *int32
     TrialConcurrency            *int32
     RequestTemplate             string
}
```

Listing 1. Morphling Programming Interface.

5 Cloud-Native Implementation

We have implemented the meta-learning algorithm as a managed configuration tuning service in Kubernetes [6], which we call Morphling. Our implementation consists of around 5k lines of Golang code and is open-sourced for public access.[1]

5.1 Programming Interface and Workflow

Programming Interface. To use Morphling for configuration tuning, users simply specify the following information through the system-provided `Experiment` interface shown in Listing 1: (1) a serving container (e.g., a Docker image) that runs an ML model, (2) the performance objective function $f(\boldsymbol{x})$,(3) tuneable configuration knobs such as resource allocations and runtime parameters,(4) the sampling algorithm (e.g., meta-learning or BO), (5) the sampling budget specifying the maximum number of sampled configurations during the search, (6) the trial concurrency specifying the maximum number of parallel trials, where a trial is an internal API that abstracts a stress-testing procedure, (7) the service request template consisting of one or more serialized client requests to query the inference service for stress test.

Workflow. Fig. 5 illustrates the system components of Morphling and their interaction workflow. To start a configuration tuning process, a user submits an *experiment* request to an *experiment controller* by making an RPC call or through

[1]Morphling is now integrated into KubeDL as a Cloud Native Computing Foundation project at https://github.com/kubedl-io/morphling.

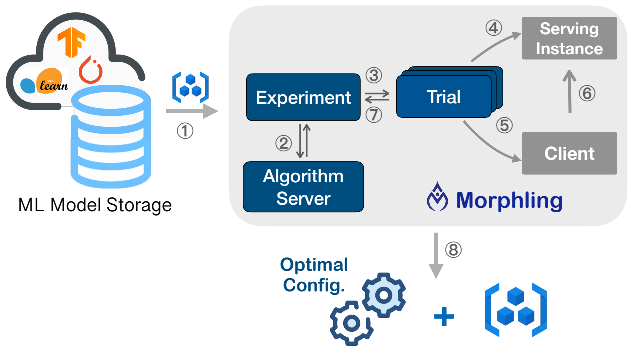


Figure 5. Morphling's implementation and workflow.

a front-end UI, specifying the serving container of the ML model, the tuneable configuration knobs, optimization objectives, and sampling budgets (①). During an experiment, Morphling iteratively communicates with an *algorithm server* which returns the next sampled configuration (②), and then starts a *trial* to evaluate that configuration (③). In each trial, a model serving instance is launched, followed by a stress test initiated by a client (④, ⑤ and ⑥). After the test completes, the measured performance (e.g., peak RPS) is stored into a database. A trial completes after all the results are sent back to the experiment controller (⑦). Morphling launches trials iteratively, until the sampling budget exhausts. The experiment hence completes, and the optimal configuration is obtained.

5.2 System Components

Controllers. Morphling defines an experiment and a trial as Kubernetes CRDs (Custom Resource Definition) [1]. For each CRD, Morphling designs a *controller* to manage its life cycle. Specifically, a controller is a long-running process that orchestrates container operations and drives the current cluster state towards a desired state. For example, the experiment controller governs the entire configuration tuning process with iterative trials; the trial controller manages low-level container behaviors, such as launching a serving container and initializing a client-side stress test. Such design automates container orchestrations and provides a simple interface to users.

Algorithm Server. Morphling trains a meta-model offline with a 2-layer neural network in TensorFlow. It uses the meta-model as the initial regression model at the beginning of an experiment and gradually refines it to navigate configuration search. Morphling implements the entire algorithm in an *algorithm server* running in a separate container. The server exposes an RPC interface through which it accepts a query from the experiment controller and returns the next configuration to it. In addition to meta-learning, the server also provides interfaces for users to implement other auto-configuration algorithms like BO and grid search.

Table 1. Open-source models used in the evaluation (42 in total), provided by the TensorFlow model zoo [8, 15].

Model Type	Model Families (# of models)
Img. Class.	ResNet (5), NASNet (2), VGG (2), Inception (2), DenseNet (1), MobileNet (2), EfficientNet (7),
Lang. Mod.	BERT (2), ALBERT (4), ELMo (1), NNLM (2), Small BERT (4), Word2vec (2), ELECTRA (2), Universal Sentence Encoder (4)

Table 2. EC2 instances used in the evaluation.

Instance Type	# of CPUs	GPU Type	$/hour
`g4dn.2xlarge`	8	T4	0.75
`p3.2xlarge`	8	Tesla V100	3.06
`g4ad.4xlarge`	16	Tesla M60	0.87
`c6g.4xlarge`	16	None	0.54

Client. Morphling provides an out-of-the-box client to stress-test an inference service under a selected configuration. For each service container, a client sends concurrent requests via REST APIs. It gradually increases the request load and reports the peak RPS without violating the target latency SLOs.

Storage Database. Morphling logs the measured configuration performance of each trial to a metric storage, currently implemented as a MySQL database container that can be accessed via an RPC interface.

6 Evaluation

In this section, we evaluate Morphling with popular open-source models in AWS EC2 and real-world inference services running in our production cluster. Our evaluations aim to answer three questions. (1) How does Morphling perform compared to existing auto-tuning solutions in terms of configuration optimality and search cost (§6.1.2 and §6.2)? (2) Can Morphling support different performance objectives (§6.1.2)? (3) How does Morphling quickly adapt to a new configuration task (§6.1.3)?

6.1 Serving Open-Source Models in EC2 Clusters

Our evaluation starts with an EC2 deployment that serves popular open-source models.

6.1.1 Methodology

Open-Source Models. Following the guideline of the MLPerf Inference benchmark [45], we choose 42 models of various sizes in 15 model families (Table 1), including image classification models like ResNet [31, 32], EfficientNet [59], and MobileNet [34, 53], and language models like BERT [23], ALBERT [42], and Universal Sentence Encoder [20]. These pre-trained models are provided by TensorFlow model zoos [8, 15]. We package both the model and the serving framework in a Docker container [3], along with an interface to configure the resources and batch size upon container launching.

Table 3. Search space for open-source models.

Configuration	Candidate choices
CPU cores	1, 2, 3, 4, 5
GPU memory	5%, 10%, 15%, 20%, 30%, 40%
Batch size	1, 2, 4, 8, 16, 32, 64, 128
GPU type	`T4`, `Tesla V100`, `Tesla M60`

Search Space. We consider four tuneable configuration knobs for a model serving container: (1) CPU cores, (2) GPU memory (in percentage of the total capacity), (3) request batch size, and (4) GPU type. For each configuration knob, we perform offline measurement to determine its search space. For example, we do not consider a configuration with > 5 CPU cores as it cannot further improve the inference RPS. Table 3 summarizes the possible choices for each configuration knob. Together, we have a total of 720 configuration options in the search space. This is considered large compared to the existing cloud configuration works, e.g., the search space of VM configurations studied in Cherrypick [18] and Scout [36] has only dozens of choices.

Objective. We set the objective of configuration tuning as to *maximize the service throughput per monetary cost*, i.e.,

$$\max_{x\in\mathcal{A}} \quad \text{RPS/Cost}. \tag{7}$$

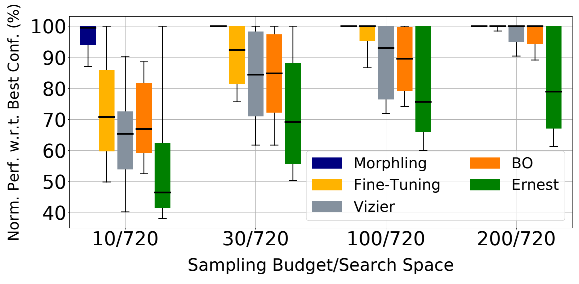
In particular, we stress-test a container's peak RPS subject to a latency SLO, set to 1 second in the experiment. To measure the monetary cost of model serving, we assume the following cost model: Cost = base cost + GPU price × GPU memory + CPU price × # of CPU cores. Table 2 compares four EC2 instances with different resources and prices, based on which we set the hourly rate of each resource as follows: base cost = 0.2 USD, CPU price = 0.02 USD, `T4` price = 0.4 USD, `M60` price = 0.4, and `V100` price = 2.6 USD.

Morphling Settings. The meta-model used in Morphling is a neural network with two hidden layers, each having 128 hidden units. Among all 42 ML models, we use 10 models for meta-training and the others for testing. For a fair and reproducible comparison, we choose 8 fixed configurations as the initial sampling points, including the maximum and the minimum values of CPU cores, GPU memory, and batch size. We use `T4` GPUs in the experiments. We find that sampling these configurations as initial points leads to the best performance for the baseline algorithms, especially BO, while Morphling is insensitive to the initial choices.

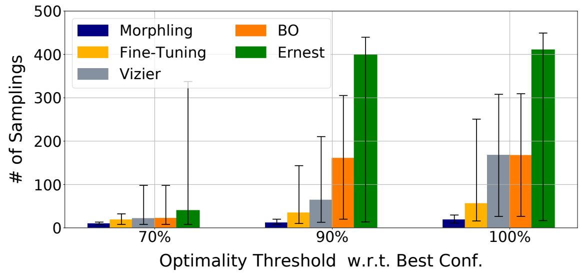
Baselines. We compare Morphling with five baseline algorithms for auto-configuration.

1) Bayesian optimization (BO): Similar to Cherrypick [18], we use a Gaussian regressor with upper confidence bound as the acquisition function.

2) Ernest [61] builds a dedicated regression model for each workload and trains it with a few samplings. We use the same neural network architecture as in Morphling, but train it for each inference service from scratch.



(a) Normalized performance of configurations returned by different approaches with varying sampling budget.



(b) Search cost incurred by different approaches to meet certain performance requirements (e.g., 70%, 90%, and 100% optimality).

Figure 6. Configuration performance and search cost of 32 opens-source models, using different tuning approaches.

3) Google Vizier [27] is a similarity-based search framework which uses a Gaussian regressor as the kernel function and employs transfer learning to accelerate a new search with well-profiled benchmarks. We use 10 ML models as the offline benchmarks. Each testing model is then represented by the data from the benchmarks, plus a Gaussian regressor that fits the testing benchmark residual.

4) Fine-Tuning is another simple, yet effective similarity-based search approach. Similar to Morphling, we offline train a regression model with 10 ML models. Yet, the objective is to simply improve the average prediction accuracy, without considering fast adaptation in the future. The trained model is then refined for a new service model.

5) Random search generates different configurations by randomly sampling the search space and takes the one with the best performance. In our implementation, the same configuration will only be sampled once, i.e., random search without replacement.

Metrics. We use two metrics in evaluation. (1) The resultant performance under a chosen configuration, defined by the objective function in Eq. (7). We report the *normalized value* with respect to the performance of the optimal configuration found by exhaustive search. (2) The search cost, measured by the number of samplings needed to find a configuration that meets a certain performance requirement, e.g., 70% of the optimum.

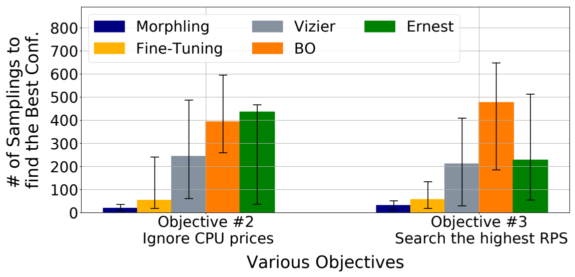


Figure 7. Morphling remains the most efficient under various objective functions. The bars measure the median search costs of tuning 32 models, and the error bars extend to the 10th and the 90th percentile.

6.1.2 Performance, Cost, and Generality

Search Quality and Search Cost. We tune the configurations of all 32 test models using Morphling and the five baseline approaches. Fig. 6a compares the normalized performance of the identified configurations under different sampling budgets, where boxes depict the 25th, 50th, and 75th percentiles of the performance of all test models, and whiskers depict the 10th and the 90th percentile. With the same sampling budget, Morphling always returns a better configuration than the five baselines. In fact, Morphling identifies the optimal configuration for all models by sampling no more than 30 configurations (less than 5% of the search space); Fine-Tuning, the second-best algorithm, requires sampling 200 configurations, yet still cannot guarantee the optimal performance for *all* models.

Fig. 6b further compares the search cost needed by different approaches to meet certain performance requirements, where the bars measure the median costs of tuning 32 models with the error bars extending to the 10th and the 90th percentile. In all cases, Morphling outperforms the other baselines: the higher the performance requirement is, the more efficient it becomes. In particular, when searching for the optimal configuration (100% optimality), Morphling is 3× efficient than Fine-Tuning (requiring a median of 54 samplings), 9.4× efficient than BO and Google Vizier, and over 22× efficient than Ernest and random search. It is worth mentioning that Ernest, though efficient in solving simpler cloud auto-tuning problems with a small configuration space like VM selection [61], falls short in navigating high-dimensional search mandated by model serving.

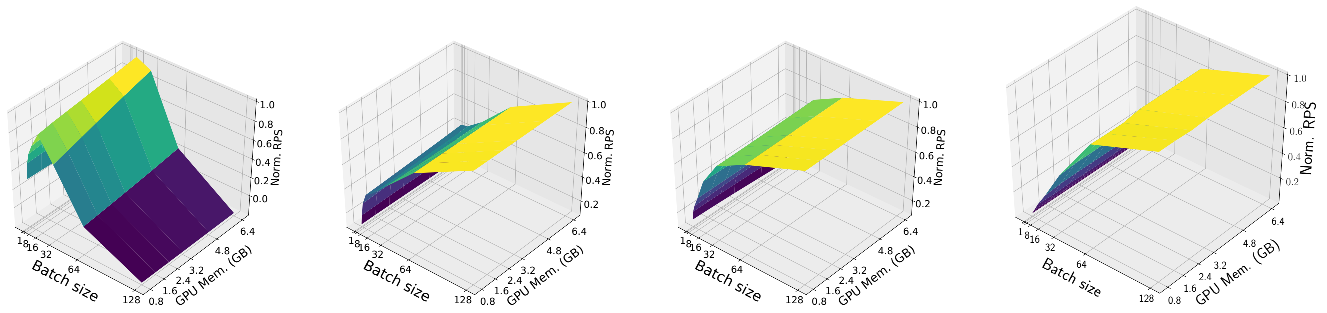
Support of Different Objective Functions. The high performance and low cost of Morphling are not tied to a particular objective function, but generally applies to a broad range of objectives. To show this, we define two tuning objectives, referred to as objectives #2 and #3 as opposed to the definition in previous experiments. In particular, objective #2 is similarly defined based on Eq. (7), except that it ignores the CPU price (CPU price = 0). This definition is well justified in production clusters as operators are mainly concerned with better utilizing high-cost GPUs. Objective #3 is set to pursue the highest RPS regardless of the monetary cost. Fig. 7 compares the search cost of Morphling and the five baselines under the two objectives. Morphling retains its advantage over the baselines, always returning the optimal configuration within 30 samplings. Compared to Fig. 6b, BO sees a sharp efficiency drop under the two new objectives, requiring a median of more than 400 samplings (55% of the search space) to find the optimal configuration. We note that objectives like searching for the highest RPS often result in a highly uneven configuration-performance plane with multiple *local optimums*, where BO can be easily stuck.

6.1.3 Microbenchmark

Fast Adaptation for New Regression Tasks. Morphling's high efficiency is attributed to its ability of quickly adapting the meta-model to a new inference service. To illustrate this, we consider tuning two configuration knobs, GPU memory and maximum batch size, for a language model `NNLM-128` [54], and depict the adaptation of the meta-model in Fig. 8. Fig. 8d shows the configuration-RPS plane manually measured for the model. During the configuration sampling process, the goal of regression is to fit this mapping plane. Figs. 8a, 8b and 8c visualize the mapping planes given by the initial meta-model θ^* (the initial regression model), the adapted model after 8 initial samplings, and that after 28 samplings, respectively. The meta-model, after samplings 8 fixed configurations in the initial stage, is quickly adapted towards the ground truth. Shortly after 28 samplings, the fine-tuned regression can accurately fit the target. This explains why Morphling can find the global optimum in a few shots. In comparison, Fig. 9 visualizes the fitting process of BO for the same model `NNLM-128` [54], where the fitted plane remain far from the ground truth after 28 samplings.

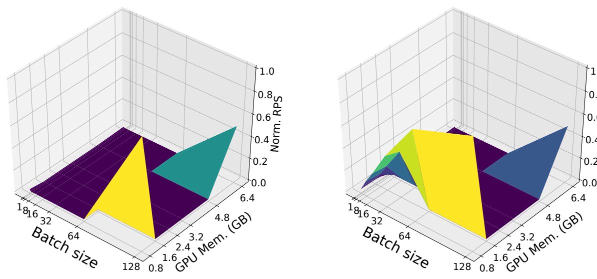
Search Path. To further explain the search path of Morphling along with fast adaptation, we illustrate the first 10 sampled configurations after the fixed initial samplings. We consider two ML models: `Universal-Sentence-Encoder` [20] (`un.se.en`) and `EfficientNetb5` [59] (`effic.5`). The tuning objective is set to Eq. (7). Exhaustive profiling shows that the optimal configurations for `effic.5` and `un.se.en` are ⟨3 CPU cores, 5% GPU memory, `V100`, batch size 8⟩ and ⟨1 CPU cores, 10% GPU memory, `T4`, batch size 128⟩, respectively.

Fig. 10a visualizes the two models' search paths in a 2-dimensional space. Both searches start at the same points (1 CPU core, batch size 16), yet expand to different paths leading to their respective optimums (marked with stars). Similar results are also shown in Fig. 10b, where Morphling quickly identifies that `T4` is most suited for `un.se.en`, and `V100` is the best fit for `effic.5`.



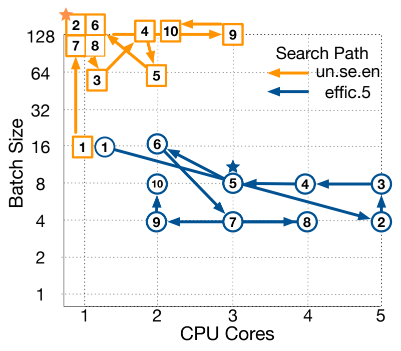
(a) Morphling: meta-model. **(b)** Morphling: 8 initial samplings. **(c)** Morphling: 8 + 20 samplings. **(d)** Ground truth.

Figure 8. An illustration of Morphling quickly adapting the meta-model to a language model NNLM-128. There are two tunable knobs, GPU memory and the maximum batch size. The RPS is normalized by the highest that the inference service can achieve.



(a) BO: 8 initial samplings. **(b)** BO: 8 + 20 samplings.

Figure 9. An illustration of BO's regression process for NNLM-128. The RPS is normalized by the maximum value.



Step	un.se.en	effic.5
1	Tesla M60	T4
2	Tesla M60	Tesla V100*
3	T4*	Tesla V100
4	T4	Tesla V100
5	T4	Tesla V100
6	T4	Tesla M60
7	T4	Tesla V100
8	Tesla M60	Tesla V100
9	T4	Tesla V100
10	T4	Tesla V100

(a) CPU cores and batch size. **(b)** GPU type.

Figure 10. Search paths of two open-source models in Morphling. Optimal configurations are marked with stars (*).

6.2 Serving Production Models in Alibaba

We have deployed Morphling in a production cluster in Alibaba to auto-tune the configurations of inference services for optimal performance.

Production Inference Services. Our evaluation includes 30 production inference services that are widely deployed to support the company's online retailing businesses. They run state-of-the-art ML models for commodity classification, production recommendation, object detection, video processing, pornography detection, etc. In total, there are 364 container instances running in the cluster. Each service container contains both an ML model and a complicated *pre/post-processing pipeline*, such as data compression, feature extraction, legality check, etc. The services are hence more demanding in CPUs than the service containers running open-source models (§6.1).

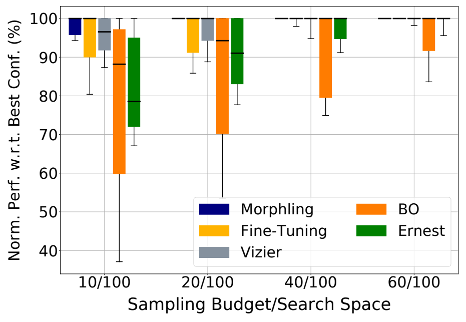
Search Space for Configuration Tuning. In our evaluation, we use T4 GPUs, which provide the best performance-cost ratio according to our experience. We use Morphling to tune three configuration knobs for each inference service: (1) CPU cores, (2) GPU memory size, and (3) GPU timeshare. As mentioned before, Alibaba has developed a GPU sharing technique that allows a GPU to be time-multiplexed by multiple containers, while ensuring a strong isolation between those containers. With this technique, GPUs can be allocated to containers the same way as CPUs. Table 4 summarizes the possible choices for each configuration knob. In total, we have 100 configuration options in the search space.

Objective. We set the same configuration tuning objective as in previous studies, that is, maximizing the service RPS per monetary cost (Eq. (7)). We assume the same cost model specified in §6.1, with a new term added to reflect the cost of GPU timeshare. Formally, we define Cost = base cost + GPU Mem. price × GPU Mem. + GPU SM price × GPU Timeshare + CPU price × # of CPU cores, where we break down the T4 price into 0.2 USD for GPU SM and 0.2 USD for GPU Memory. All the other settings remain the same as in §6.1.

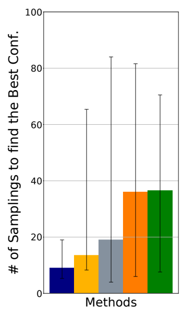
Algorithm Settings and Metrics. We use five inference services, three for image classification and two for video processing, as the training set for a tuning algorithm. The other 25 services are used for evaluation. Each tuning algorithm

Table 4. Configuration search space for production services.

Configuration	Candidate Choices
CPU cores	2, 4, 6, 8
GPU memory	12.5%, 25%, 37.5%, 50%, 62.5%
GPU timeshare	20%, 40%, 60%, 80%, 100%



(a) The normalized performance of the identified configurations for the evaluated inference services under varying sampling budgets.



(b) Search costs to find the optimal configurations.

Figure 11. Evaluations of search quality and search costs of production inference services. (a) Boxes depict the 25th, 50th, and 75th percentiles, respectively; whiskers depict the 10th and the 90th percentiles, respectively. (b) Bars depict the median; error bars measure the 10th and the 90th percentile.

initially samples four fixed configurations covering the maximum and the minimum values of both CPU cores and GPU memory, where the GPU timeshare is set to 20%.[2] For each inference service tuned by an algorithm, we normalize the measured performance by the optimum found by exhaustive search.

Evaluation Results. Fig. 11a compares the normalized performance of the configurations recommended by different algorithms for the 25 inference services under varying sampling budgets. Fig. 11b further compares the search costs required by those algorithms to find the optimal configurations. Morphling leads the five baselines in both the configuration performance and the search cost. In particular, Morphling identifies the optimal configurations with a median of 9 samplings and a maximum of 19, much more efficient than the baselines, among which Fine-Tuning is a front-runner, followed by Google Vizier, BO, Ernest, and random search. This result is in line with the previous evaluations (§6.1). For Morphling, the search cost of tuning production services (a median of 9 samplings out of 100 options) is slightly higher than tuning open-source models (a median of 18 samplings out of 720 options). This is expected as the latter has more configuration knobs and thus a larger search space, for which meta-learning usually exhibits a higher performance advantage than the existing search approaches.

Overhead. In Morphling, the configuration tuning overhead mainly comes from the trials (§5.1), each taking around 10-15 minutes, including launching the service and client containers, stress-testing the peak RPS, and results collection(Fig. 5). Among these operations, service launching is usually the most time-consuming, due to the complex deployment dependencies in the production environments. In comparison, the computation time for meta-model adaptation is negligible, which takes several seconds to complete. As for meta-model training, it usually converges within a few thousands of iterations in less than 10 minutes. Note that such overhead is offline as the meta-model only needs to be trained once, which can then be reused to adapt to a new service online via few-shot learning.

7 Discussion

Application to Other Configuration Problems. We find that the meta-learning based search approach is not limited to tune inference services, but can also be extended to other cloud configuration problems. In fact, with some modifications, we have successfully used Morphling to tune configuration parameters and resource allocations of cloud storage services like Redis [13]. Having said that, meta-learning is not a good fit for problems where the tuneable knobs vary in different compute tasks, or the knowledge about one task cannot be easily transferred to another. For example, when tuning neural network architectures [37, 58, 71, 72], hyper-parameters like the cell structure can be potentially generalized between datasets, but the network's width and depth critically depend on the current training task. For these hyper-parameters, knowledge transfer offers little help to accelerate the tuning process [37, 72].

Generalizability of the Meta-Model. One common concern about meta-learning is whether the meta-model generalizes to diverse inference services. Fortunately, we find that the identified hardware and runtime configurations (e.g. CPU cores and GPU memory) have a rather stable performance impacts to a broad range of inference services. For example, both image classification and language models can be well fitted by adapting a common meta-model, although their sensitivities to those configurations may differ. In case that a new configuration knob other than the identified ones needs to be tuned, one can simply re-train the meta-model over a small number of services, which can complete in a short period of time (e.g., less than ten minutes as mentioned in §6.1).

Other Important Model Serving Configurations. In addition to the identified hardware configurations, resources like RAM (main memory) and disk storage can also affect the quality of inference services. Yet, these resources are usually over-provisioned in a machine. For example, in EC2,

[2]These fixed initial points lead to the optimal performance for the five baselines, while Morphling is insensitive to the initial choices.

a `g4dn.4xlarge` instance provides one `T4` GPU along with 64 GB RAM and 225 GB storage, while a model serving container typically requires only several GBs of RAM and storage. Such resource over-provisioning is also found in Alibaba's production clusters [66]. We therefore do not tune their allocations. Previous work [68] also indicates that a well-tuned degree of parallelism (i.e., the number of threads) indirectly improves the quality of ML inference services, as it enables concurrent request processing, thus fully utilizing the CPUs with pipelines. However, we observe no noticeable performance difference when configuring a different number of serving threads in our experiments. We also note that network and I/O bandwidth can have significant performance impact to inference services. We choose not to include them because their allocations cannot be easily enforced at container level in production clusters and public clouds like EC2, though our algorithm can easily include them as another configuration knobs for auto-tuning.

8 Conclusion

In this paper, we presented Morphling, a fast, near-optimal auto-configuration framework for cloud-native model serving. We first identified a number of important configuration knobs that critically determine the performance and cost of an inference service, such as CPU cores, GPU memory, GPU timeshare, GPU type, and batch size. We showed that there is a general configuration-performance trend in a broad range of ML models. Based on this observation, we proposed to automatically tune the configuration of an inference service using meta-learning, which we have implemented in Morphling. Morphling trains a meta-model offline to capture the general performance trend under varying configurations. The meta-model is then used as an initial regression model to direct configuration search for a new inference service. Morphling iterates between fitting the model and using it to determine which configuration to explore, until the sampling budget runs out. We evaluated Morphling with popular open-source models and Alibaba's production inference services. Evaluation results show that Morphling supports various tuning objectives, quickly identifying the optimal configuration for a new inference service with much smaller sampling overhead than the existing auto-configuration approaches.

Acknowledgement

We thank our shepherd Alekh Jindal and the anonymous reviewers for their valuable comments that help improve the quality of this work. We are also grateful to the colleagues from the Alibaba Machine Vision Application Platform for providing the production service data and setting up the experimental environments. This research was supported in part by RGC RIF grant R6021-20, and RGC GRF grants under the contracts 16213120, 16206417 and 16207818.

References

[1] 2021. Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.
[2] 2021. Deliver high performance ML inference with AWS Inferentia. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf.
[3] 2021. Docker. https://www.docker.com.
[4] 2021. Httperf. https://github.com/httperf/httperf.
[5] 2021. Jmeter. https://jmeter.apache.org/.
[6] 2021. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.
[7] 2021. Machine Learning on AWS. https://aws.amazon.com/machine-learning.
[8] 2021. Module: Tensorflow Keras Applications. https://www.tensorflow.org/api_docs/python/tf/keras/applications.
[9] 2021. NVIDIA Data Center Deep Learning Product Performance. https://developer.nvidia.com/deep-learning-performance-training-inference.
[10] 2021. NVIDIA TensorRT Inference Server. https://github.com/triton-inference-server/server.
[11] 2021. NVIDIA TESLA M60 GPU ACCELERATOR. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/nvidia-m60-datasheet.pdf.
[12] 2021. Nvidia Virtual GPU Technology. https://www.nvidia.com/en-us/data-center/virtual-gpu-technology/.
[13] 2021. Redis: an open source, in-memory data structure store. https://redis.io.
[14] 2021. Siege. https://www.joedog.org/siege-home/.
[15] 2021. TensorFlow Hub. https://tfhub.dev/.
[16] 2021. TensorFlow Serving for model deployment in production. https://www.tensorflow.org/serving/.
[17] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. 2014. Laser: A scalable response prediction platform for online advertising. In *Proc. ACM WSDM, 2014.*
[18] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. USENIX, 2017.*
[19] Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade.* Springer, 421–436.
[20] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Céspedes, Steve Yuan, Chris Tar, et al. 2018. Universal sentence encoder. *arXiv preprint arXiv:1803.11175* (2018).
[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI, 2017.*
[22] Brian Dalessandro, Daizhuo Chen, Troy Raeder, Claudia Perlich, Melinda Han Williams, and Foster Provost. 2014. Scalable hands-free transfer learning for online advertising. In *Proc. ACM SIGKDD, 2014.*
[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[24] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proc. ACM EuroSys, 2012.*
[25] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proc. PMLR ICML, 2017.*
[26] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

[27] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proc. ACM SIGKDD, 2017*.

[28] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proc. ACM/IFIP/USENIX Middleware, 2017*.

[29] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proc. USENIX OSDI, 2020*.

[30] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *Proc. IEEE HPCA, 2018*.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proc. IEEE/CVF CVPR, 2016*.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *Proc. ECCV, 2016*.

[33] Trong Nghia Hoang, Quang Minh Hoang, Ruofei Ouyang, and Kian Hsiang Low. 2018. Decentralized high-dimensional Bayesian optimization with factor graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[34] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[35] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. 2018. Arrow: Low-level augmented bayesian optimization for finding the best cloud vm. In *Proc. IEEE ICDCS, 2018*.

[36] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent W Freeh. 2018. Scout: An experienced guide to find the best cloud configuration. *arXiv preprint arXiv:1803.01296* (2018).

[37] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. 2016. Deep networks with stochastic depth. In *Proc. ECCV, 2016*.

[38] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041* (2018).

[39] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[40] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *Proc. ACM SIGCOMM, 2018*.

[41] Jiho Kim, Jehee Cha, Jason Jong Kyu Park, Dongsuk Jeon, and Yongjun Park. 2018. Improving GPU multitasking efficiency using dynamic resource sharing. *IEEE Comput. Archit.* 18, 1 (2018), 1–5.

[42] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. *arXiv preprint arXiv:1909.11942* (2019).

[43] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the black box of machine learning prediction serving systems. In *USENIX OSDI, 2018*.

[44] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. 2017. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835* (2017).

[45] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. 2020. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro, 2020* 40, 2 (2020), 8–16.

[46] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[47] Valerio Perrone, Huibin Shen, Matthias Seeger, Cedric Archambeau, and Rodolphe Jenatton. 2019. Learning search spaces for bayesian optimization: Another view of hyperparameter transfer learning. *arXiv preprint arXiv:1909.12552* (2019).

[48] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. 2018. Scanner: Efficient video analysis at scale. *ACM Trans. Graph., 2018* 37, 4 (2018).

[49] Santu Rana, Cheng Li, Sunil Gupta, Vu Nguyen, and Svetha Venkatesh. 2017. High dimensional Bayesian optimization with elastic Gaussian process. In *Proc. PMLR ICML, 2017*.

[50] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proc. USENIX ATC, 2021*.

[51] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In *Proc. ACM EuroSys, 2020*.

[52] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[53] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proc. IEEE/CVF CVPR, 2018*.

[54] Yongzhe Shi, Wei-Qiang Zhang, Meng Cai, and Jia Liu. 2014. Efficient one-pass decoding with NNLM for speech recognition. *IEEE Signal Process. Lett.* 21, 4 (2014), 377–381.

[55] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[56] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *arXiv preprint arXiv:1206.2944* (2012).

[57] Richard Socher. 2014. *Recursive deep learning for natural language processing and computer vision*. Citeseer.

[58] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proc. IEEE/CVF CVPR*.

[59] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR.

[60] Takeshi Teshima, Issei Sato, and Masashi Sugiyama. 2020. Few-shot domain adaptation by causal mechanism transfer. In *Proc. PMLR ICCV, 2020*.

[61] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proc. USENIX NSDI, 2016*.

[62] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: machine learning as an analytics service system. *VLDB Endowment, 2018* 12, 2 (2018).

[63] Yu-Xiong Wang and Martial Hebert. 2016. Learning to learn: Model regression networks for easy small sample learning. In *Proc. Springer ECCV, 2016*.

[64] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of service support for fine-grained sharing on GPUs. In *Proc. ACM/IEEE ISCA, 2017*.

[65] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, Nando De Freitas, et al. 2013. Bayesian Optimization in High Dimensions via Random Embeddings.. In *Proc. IJCAI, 2013*.

[66] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Chen Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large Heterogeneous GPU Clusters. In *Proc. USENIX NSDI, 2022.*

[67] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proc. ACM SoCC, 2017.*

[68] Feng Yan, Olatunji Ruwase, Yuxiong He, and Evgenia Smirni. 2016. SERF: efficient scheduling for fast deep neural network serving via judicious parallelism. In *Proc. IEEE SC, 2016.*

[69] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610* (2019).

[70] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proc. USENIX ATC, 2019.*

[71] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[72] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proc. IEEE/CVF CVPR, 2018.*

[73] Corey Zumar. 2018. InferLine: ML Inference Pipeline Composition Framework. (2018).