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\times-22\times, quickly converging to the optimal configuration by sampling only 30 candidates in a large search space consisting of 720 options.

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 10x 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.
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 on 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 10x 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 3x, 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 [35] 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

\[ \mathbf{x} = \{x_1, x_2, \ldots, x_M\} \]

where each \( x_i \) has a discrete search space consisting of \( n_i \) candidates. Our goal is to find the optimal configuration \( \mathbf{x}^* \) that maximizes the objective function \( f(\cdot) \) defined by the operator, i.e.,

\[ \mathbf{x}^* = \arg \max_{\mathbf{x} \in \mathcal{A}} f(\mathbf{x}), \]  

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 \( f(\mathbf{x}) \) 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
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

![Figure 3. Normalized RPS under different configurations of batch size and GPU memory.](image)

![Figure 4. Normalized RPS under different configurations of CPU cores and GPU memory.](image)
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 autoconfiguration 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 \( T \) with a mapping function \( f_\theta(x) \) that predicts the service performance for an input configuration \( x \), where \( \theta \) is the parameterized model tuned for task \( T \). The loss function \( L_T \) is defined as the mean squared error (MSE) between the predicted performance and the real performance \( y \), i.e.,

\[
L_T(f_\theta) = \sum_{x,i,y} ||f_\theta(x) - y||^2_2. \tag{2}
\]

In a \( K \)-shot regression, model \( \theta_i \) is trained with \( K \) sampled input-output pairs \( D = \{x_j, y_j| j = 1, 2, \ldots, K\} \); the objective is to minimize the loss \( L_T(f_\theta) \). 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 \( T = \{T_1, T_2, \ldots, T_N\} \) that share common input-output mapping features. Let \( \theta^m \) be the meta-model trained in the first stage. Given a new task \( T_i \), MAML adapts \( \theta^m \) to a fine-tuned model \( \theta_i \) by iteratively updating it with \( K \) newly sampled data points \( D_{i|K|} = \{x_j, y_j| j = 1, 2, \ldots, K\} \), using stochastic gradient descent (SGD) [19], i.e.,

\[
\theta_i = \theta^m - \alpha \nabla_{\theta^m} L_{T_i}(f_{\theta^m}), \tag{3}
\]

where \( \alpha \) 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 later SGD process can make a rapid progress without over-fitting. Therefore, the meta-model \( \theta^m \) is trained to optimize \( f_\theta \) over tasks sampled from \( T \), i.e.,

\[
\theta^m \leftarrow \theta^m - \beta \nabla_{\theta^m} \sum_{T \in T} L_{T}(f_\theta), \tag{4}
\]

where \( \beta \) is the learning step in the meta-training stage, and \( \theta^m \) 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 \( \theta^m \) 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 \( \theta^m \) 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 \( \theta^m \) 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
where $\delta$ is a pre-defined weight knob which is usually a small constant. Algorithm 2 details the search process with the meta-model.

Algorithm 2: SMBO with Meta Model

**Input**: A new regression task $T_i$, learning rates $a$, sampling budget $K$, meta model $\theta^m$.

**Output**: The optimal configuration $x^*$

1. Initialize $\theta_i \leftarrow \theta^m$, and newly-sampled data set $D \leftarrow \emptyset$.
2. For $k = 1, 2, \ldots, K$
3. Update regression model $\theta_i' = \theta_i - a \nabla_{\theta_i} L_{T_i}(f_{\theta_i})$.
4. For all $x \in A$
5. Calculate the $f_{\theta_i}(x)$ and Acq($f_{\theta_i}(x)$) using Eq. (6).
6. $x^k \leftarrow \arg \max_{x \in A \cap x \notin D} \text{Acq}(f_{\theta_i}(x))$.
7. Estimate $y^k$ for $x^k$.
8. $D \leftarrow D \cup (x^k, y^k)$, $\theta_i \leftarrow \theta_i'$.
9. $x^* \leftarrow \arg \max_{x \in D} y^k$.

The maximum number of configurations that the algorithm can explore. Let $\theta_i$ be the refined model for $T_i$. Given sampling budget $K$, our algorithm sequentially explores the next configuration by predicting the performance $f_{\theta_i}(x^k)$ for all candidate configurations $x^k$ 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 $x$, let $f_{\theta_i}(x)$ be the performance prediction made by model $\theta_i$. Upon sampling, the model is updated to $\theta_i' = \theta_i - a \nabla_{\theta_i} L_{T_i}(f_{\theta_i})$, and the new prediction becomes $f_{\theta_i'}(x)$. We define the confidence of $f_{\theta_i'}(x)$ as

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

That is, a lower Conf(.) indicates a higher confidence. Intuitively, if two sequential regression models $\theta_i$ and $\theta_i'$ 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:

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

where $\delta$ 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.

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 $f(x)$, (3) tuning 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.
a front-end UI, specifying the serving container of the ML model, the tuneable configuration knobs, optimization objectives, and sampling budgets (③). 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].

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

Table 2. EC2 instances used in the evaluation.

<table>
<thead>
<tr>
<th>Instance Type</th>
<th># of CPUs</th>
<th>GPU Type</th>
<th>$/hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>g4dn.2xlarge</td>
<td>8</td>
<td>Tesla V100</td>
<td>3.06</td>
</tr>
<tr>
<td>g4ad.4xlarge</td>
<td>16</td>
<td>Tesla V100</td>
<td>0.87</td>
</tr>
<tr>
<td>c6g.4xlarge</td>
<td>16</td>
<td>None</td>
<td>0.54</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Candidate choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU cores</td>
<td>1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>GPU memory</td>
<td>5%, 10%, 15%, 20%, 30%, 40%</td>
</tr>
<tr>
<td>Batch size</td>
<td>1, 2, 4, 8, 16, 32, 64, 128</td>
</tr>
<tr>
<td>GPU type</td>
<td>T4, Tesla V100, Tesla M60</td>
</tr>
</tbody>
</table>

Search Space. We consider four tunable 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.,

$$\max_{x \in \mathcal{X}} \frac{\text{RPS}}{\text{Cost}}$$

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: \(\text{Cost} = \text{base cost} + \text{GPU price} \times \text{GPU memory} + \text{CPU price} \times \# \text{ of CPU cores}\). 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.

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.
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 $\theta^*$ (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.
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

![Figure 8](image1.png)

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.

![Figure 9](image2.png)

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

![Figure 10](image3.png)

Figure 10. Search paths of two open-source models in Morphling. Optimal configurations are marked with stars (*)
Table 4. Configuration search space for production services.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Candidate Choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU cores</td>
<td>2, 4, 6, 8</td>
</tr>
<tr>
<td>GPU memory</td>
<td>12.5%, 25%, 37.5%, 50%, 62.5%</td>
</tr>
<tr>
<td>GPU timeshare</td>
<td>20%, 40%, 60%, 80%, 100%</td>
</tr>
</tbody>
</table>

(a) The normalized performance of the identified configurations for the evaluated inference services under different 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 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%. 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 model 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,
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


