Provisioning of Kubernetes Clusters for Task-Based Python Applications

Andrey Nagiyev*†, Enes Bajrovic*, Siegfried Benkner*
*Faculty of Computer Science, University of Vienna, Vienna, Austria
†Doctoral School Computer Science, University of Vienna, Vienna, Austria
andrey.nagiyev@univie.ac.at, enes.bajrovic@univie.ac.at, siegfried.benkner@univie.ac.at

***Abstract*—We present Python-to-Kubernetes (PTK), a framework that automates the provisioning and deployment of task-based Python applications on Kubernetes. PTK introduces compact source-code annotations for tasks, resource needs, grouping, and data-size hints. From these annotations, it provisions an application-specific cluster, builds container images, generates manifests, and selects the data-transfer mechanism based on placement. A scoring-based mapping co-locates bandwidth-heavy neighbors to reduce cross-node traffic and right-sizes nodes after placement. On a six-task machine learning (ML) ResNet50 image-classification pipeline (ImageNet 5%/10% subsets), PTK achieves up to 6.43× faster runtime and 7.26× lower cost per run than the Kubernetes Default Scheduler, with higher CPU/memory utilization and fewer/smaller nodes. These results indicate that lightweight annotations plus application-aware provisioning can substantially improve price-performance for Kubernetes-based ML pipelines.**

***Index Terms*—cluster provisioning, Kubernetes, resource management, source code annotations, task-based programming**

I. Introduction

Modern ML pipelines and scientific workflows increasingly rely on heterogeneous clusters with CPUs and GPUs, yet deploying and tuning such applications on Kubernetes [1] remains complex: developers must containerize tasks, hand-craft manifests, choose node families/zones, and reason about data movement across Pods and nodes.

PTK (i) introduces compact Python annotations for tasks, resource needs, grouping, and data-size hints; and (ii) provisions an application-specific heterogeneous cluster prior to deployment. PTK then maps tasks to nodes with awareness of inter-task bandwidth and cost. Beyond placement, PTK automatically builds container images and generates Kubernetes manifests. Because task outputs must be serialized across Pods, PTK also selects the data-transfer mechanism by placement. We demonstrate PTK on a TensorFlow-based image-classification pipeline using ImageNet [2]–[5].

We contribute: (1) a minimal annotation interface for task-based Python; (2) a pre-deployment provisioning-and-mapping procedure that sizes nodes and co-locates high-bandwidth neighbors; (3) a prototype that emits images, manifests, and volumes end-to-end; and (4) an evaluation indicating improved utilization, runtime, and cost versus the Kubernetes Default Scheduler.

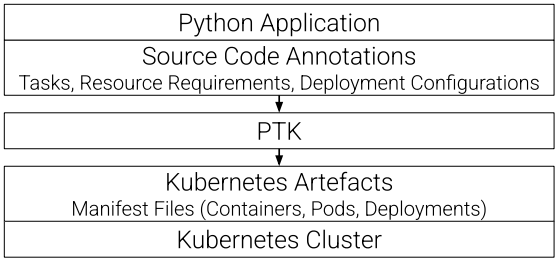

Fig. 1. Overview of the PTK framework.

II. PTK Overview

Applications are expressed as Python tasks whose outputs feed the inputs of downstream tasks (value semantics). PTK serializes arguments and return values with JSON or NumPy as appropriate and supports Keras/TensorFlow model artifacts (e.g., HDF5). From these annotations (below), PTK later builds container images, generates Kubernetes manifests, and wires volumes according to placement [6]. Figure 1 summarizes the PTK workflow and how annotations drive provisioning, mapping, and manifest generation.

Each function decorated with `@task` is a self-contained unit with value semantics: arguments are serialized inputs; the return value is the serialized output. PTK supports standard Python types (JSON), NumPy arrays (`.npy`), and Keras/TensorFlow artifacts (e.g., HDF5 model files). The `cpu_requests`, `ram_requests`, and optional `gpu_requests` describe resource needs used by the provisioning and mapping pipeline. The optional `input_data`/`output_data` provide size hints (e.g., 8 GiB) that steer data-locality decisions. By default, PTK builds one container per task and one Pod per container unless the user groups tasks or PTK groups data-heavy neighbors automatically. Users can enforce grouping (`@container`, `@pod`); PTK may also group data-heavy neighbors on the same node into one Pod to enable shared-memory transfer (`emptyDir`). Otherwise, PTK injects *PersistentVolumeClaims* (PVCs): a local PVC for same-node Pods and an NFS-backed PVC across nodes.

We build an ML ImageNet pipeline; the task roles and resource hints are as follows (see Listing 1 and Figure 2).

The heaviest edge is from `test_prepare` to `evaluate` (16 GiB). PTK uses this hint to co-locate these neighboring

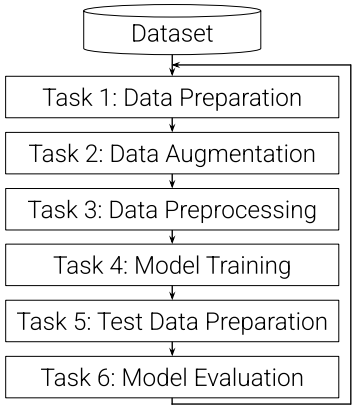

Fig. 2. Task-based ML pipeline for image classification. The loop indicates that the pipeline is deployed as a continuously running application service, with model training triggered by the user.

tasks within a single Pod (on the same node), switching from PVC/NFS transfers to in-memory exchange (`emptyDir`).

III. Provisioning and Mapping

PTK provisions an application-specific cluster before deployment and maps tasks with awareness of resource needs and data movement.

A. Cluster Provisioning Options

Users specify the cloud platform, node family (e.g., `n1`), optional zone, and optional `num_nodes`. PTK currently targets GCP and is extensible to other clouds. If the zone is omitted, PTK selects a low-cost zone within the chosen family based on standard-node pricing and then enumerates node types (marking GPU-capable variants).

```
python tasks.py --platform=gcp --node_families=n1 \
  --zone=us-west1-c --num_nodes=3
```

PTK retrieves the hourly price of the target GPU (e.g., T4) in the selected zone to compute the GPU/node price ratio, which anchors GPU scoring to node economics. A margin is reserved for Kubernetes overhead; PTK never assigns 100% of advertised capacities.

B. Scoring System

Nodes and tasks are ranked with an additive model; after mapping, a data term penalizes inter-node traffic. Table I summarizes the scores; PTK selects the cluster that minimizes the sum of node and data scores.

C. Algorithm 1: Node Selection and Task Mapping

PTK evaluates candidate cluster sizes $i = 1..t$ for t tasks. For a fixed i, it first sorts tasks by their task score (from requests of CPU/memory/GPU) and instantiates i nodes as the smallest standard machine types in the chosen family. It then seeds the top-i tasks, one per node, upgrading a node only if necessary to satisfy all three resource dimensions (CPU, memory, GPU) after reserving a small overhead for Kubernetes. GPU-requiring tasks force a GPU-capable shape and the required device count; if the family/zone cannot provide that shape, the current candidate cluster is deemed infeasible and the algorithm moves on to size $i+1$.

After seeding, the algorithm performs a greedy fill of the remaining tasks. For each task in score order, it computes the residual capacity of each node (node score minus the sum of scores of tasks already placed on that node) and attempts to place the task on the node with the largest residual capacity that also individually meets CPU, Memory, and GPU constraints. If no current node can host the task, PTK minimally upgrades exactly one node: it chooses the node that requires the smallest score increase (difference between the upgraded node's score and its current score) to satisfy the task's requirements, preferring shapes that preserve GPU headroom when the remaining task list contains GPU jobs.

If all tasks fit, the mapping for size i is recorded as

```python
from ptk import task

@task(name='prepare', cpu_requests='10',
  ram_requests='12Gi', input_data={'size':
  '8Gi'}, output_data={'size': '8Gi'})
def prepare():
    # Task 1: Prepare raw data
    return prepare_result

@task(name='augment', cpu_requests='20',
  gpu_requests='1', ram_requests='16Gi',
  output_data={'size': '8Gi'})
def augment(prepare_result):
    # Task 2: Apply data augmentation
    return augment_result

@task(name='preprocess', cpu_requests='20',
  ram_requests='16Gi', output_data={'size':
  '8Gi'})
def preprocess(augment_result):
    # Task 3: Preprocess augmented data
    return preprocess_result

@task(name='train', cpu_requests='32',
  gpu_requests='2', ram_requests='20Gi',
  output_data={'size': '2Gi'})
def train(preprocess_result):
    # Task 4: Train the model
    return train_result

@task(name='test_prepare', cpu_requests='16',
  ram_requests='12Gi', output_data={'size':
  '16Gi'})
def test_prepare(train_result):
    # Task 5: Prepare test data
    return test_prepare_result

@task(name='evaluate', cpu_requests='12',
  ram_requests='8Gi', output_data={'size': '2Gi'})
def evaluate(test_prepare_result):
    # Task 6: Evaluate the model
    return evaluate_result

def main():
    prepare_res = prepare()
    augment_res = augment(prepare_res)
    preprocess_res = preprocess(augment_res)
    train_res = train(preprocess_res)
    test_prep_res = test_prepare(train_res)
    evaluate_res = evaluate(test_prep_res)
```

Listing 1. ML pipeline tasks annotated with PTK

TABLE I
PTK SCORING SYSTEM

Score Type	Scoring Formula
Memory	1 point per GiB (rounded up if necessary)
CPU	#vCPUs × memory-to-CPU ratio
GPU	#GPUs × GPU-to-node price ratio × smallest node score
Node	Memory score + CPU score + GPU score
Task	Memory score + CPU score + GPU score
Data	Node score × inter-node data (GiB)
Cluster	Sum of all Node and Data scores

Algorithm 1: Node Selection and Task Mapping

Input:
$T = \{t_1, t_2, \ldots, t_t\}$: List of t tasks ordered by descending task scores
$C = \{c_1, c_2, \ldots, c_t\}$: Set of t candidate clusters, where each cluster $c_i = \{n_1, n_2, \ldots, n_i\}$ is a set of i nodes
Output:
$C' = \{c'_i \mid s \le i \le t\}$: Subset of feasible cluster configurations $C' \subseteq C$, where $s \ge 1$ is the index of the cluster with the smallest number of nodes. Each $c'_i = \{n_1, n_2, \ldots, n_i\}$ represents a cluster of i nodes. A unique subset of the tasks in T is assigned to each node so that no task remains unassigned.

foreach *candidate cluster* $c_i \in C$ **do**
  `// 1. Assign first i tasks`
  for $j = 1$ **to** i **do**
    if *task* t_j *fits on node* n_j **then**
      Assign task t_j to node n_j;
    else if *upgrading node* n_j *allows* t_j *to fit* **then**
      Upgrade node n_j to fit t_j;
      Assign task t_j to node n_j; **break**;
  `// 2. Assign remaining tasks`
  for $j = i + 1$ **to** t **do**
    Sort nodes $n_1, \ldots, n_i$ by remaining available capacity (descending);
    foreach *node* n_k *in the sorted list* **do**
      if *task* t_j *fits on node* n_k **then**
        Assign task t_j to node n_k; **break**;
    if *task* t_j *is not assigned* **then**
      Select node n_k with the smallest required score increase to host t_j;
      Upgrade node n_k to fit t_j;
      Assign task t_j to node n_k;
  if *all tasks in* T *are successfully mapped to* c_i **then**
    if C' *is empty* **then** $s \leftarrow i$;
    Add c_i to C';

feasible; otherwise that size is rejected. The smallest i that yields a feasible mapping defines the feasibility frontier s, and all feasible mappings for sizes $i \ge s$ are retained for the locality-refinement stage (Algorithm 2). Node sizes are not reduced in Algorithm 1 (to avoid oscillations); right-sizing occurs only after locality optimization. In practice (e.g., the six-task ImageNet pipeline), this procedure quickly converges: GPU-heavy tasks seed GPU nodes, CPU/memory-heavy tasks gravitate to the nodes with the most residual capacity, and infeasible sizes are skipped early without exhaustive search.

D. Algorithm 2: Reducing Inter-node Data Transfers

Given each feasible $c_i \in C'$, PTK improves locality without increasing node sizes by first performing direct relocations. For each task, it ranks candidate destination nodes by the total data volume exchanged with tasks already on that node and attempts a move only when the destination has sufficient

Algorithm 2: Reducing Inter-node Data Transfers

Input:
$T = \{t_1, t_2, \ldots, t_t\}$: List of t tasks ordered by descending task scores
$C' = \{c'_i \mid s \le i \le t\}$: Set of feasible clusters, where each cluster $c'_i = \{n_1, \ldots, n_i\}$ has tasks already mapped to nodes
Output:
Optimized cluster configuration c^* from C' with reduced inter-node data transfers

foreach *feasible cluster* $c'_i \in C'$ **do**
  `// 1. Direct Task Relocation`
  foreach *task* $t \in T$ **do**
    Let n_{curr} be the node currently hosting t;
    Sort nodes in c'_i by the volume of data transferred with t (descending);
    foreach *node* n_j *in the ranked list preceding* n_{curr} **do**
      if t *fits on node* n_j ***and*** n_{curr} *hosts other tasks after relocation* **then**
        Relocate task t to node n_j; **break**;
  `// 2. Pairwise Task Swap`
  repeat
    improvement $\leftarrow$ **False**;
    foreach *pair of tasks* (t_x, t_y) *assigned to different nodes* **do**
      if *swapping* (t_x, t_y) *reduces total inter-node data transfer* ***and*** *both nodes have sufficient resources* **then**
        Swap tasks t_x and t_y;
        improvement $\leftarrow$ **True**;
  until *no further improvement*
  `// 3. Downgrade Nodes`
  foreach *node* $n_j \in c'_i$ **do**
    Downgrade n_j by reclaiming unused resources;
  Compute total cluster score s_i for c'_i;
Select cluster c^* with the lowest score;

residual CPU, memory, and (if required) GPU capacity and the source will still host at least one task; passes repeat until no relocation is possible. It then considers pairwise swaps across nodes and accepts a swap only if both tasks fit after the exchange and the cluster's data term (sum over nodes of node score times inter-node GiB) strictly decreases; the swap phase also iterates to a fixed point. Finally, PTK groups dependent tasks that end up on the same node into one Pod to enable in-memory exchange and right-sizes nodes by reclaiming unused CPU and memory (GPU counts preserved), recomputes scores, and selects the lowest-scoring cluster for deployment.

IV. EVALUATION

A. Workloads and Datasets

We create a six-task ResNet50 pipeline [2], [5] on ImageNet [3], [4]. Two subsets are used: 5% (64,033 images; 7.28 GiB) and 10% (128,066 images; 14.56 GiB). The pipeline tasks and their resource requests are summarized in Table II.

B. Deployment Scenarios

We compare three mechanisms (see Figure 3): (i) PTK/Isolated – PTK provisioning without grouping (each task in its own Pod; Pods communicate via PVC); (ii) PTK/Grouped – PTK provisioning with automatic grouping of connected tasks on the same node into one Pod (enables in-Pod `emptyDir`); (iii) Kubernetes Default Scheduler – default scheduler on the same node types PTK provisioned for (ii); if it cannot place all Pods, an extra node is added as needed.

TABLE II
RESOURCE REQUESTS FOR EACH ML PIPELINE TASK

Task #	Requests							
	Experiment 1				Experiment 2			
	CPU	GPU	RAM	Output	CPU	GPU	RAM	Output
Task 1	10.0	0.0	12.0	8.0	15.0	0.0	18.0	16.0
Task 2	20.0	1.0	16.0	8.0	30.0	2.0	24.0	16.0
Task 3	20.0	0.0	16.0	8.0	30.0	0.0	24.0	16.0
Task 4	32.0	2.0	20.0	2.0	48.0	4.0	30.0	2.0
Task 5	16.0	0.0	12.0	16.0	24.0	0.0	18.0	16.0
Task 6	12.0	0.0	8.0	2.0	18.0	0.0	12.0	2.0

Note: CPU values denote vCPUs (cores); GPU values denote the number of devices and are set to 0.0 if unused. Memory and Output are reported in GiB.

TABLE III
MAIN RESULTS ACROSS THREE SCENARIOS

Metric	5% subset			10% subset		
	PTK Isolated	PTK Grouped	K8s Default	PTK Isolated	PTK Grouped	K8s Default
Runtime [h]	1.70	0.28	1.80	2.65	0.96	2.76
Hourly cost [$/h]	6.05	6.05	6.81	9.07	9.07	13.51
Cost per run [$]	10.31	1.69	12.27	24.00	8.73	37.36
CPU utilization	0.97	0.97	0.85	0.98	0.98	0.71
Memory utilization	0.82	0.94	0.46	0.85	0.96	0.31
GPU utilization	0.60	0.60	0.60	0.75	0.75	0.50

C. Metrics

We report execution time (pipeline makespan per run), cost per run and hourly cluster cost (derived from provisioned node types), and resource utilization (CPU/Memory/GPU); values closer to 1 indicate better utilization.

D. Results

For the 5% subset, PTK Grouped provisions two nodes: Node 1 (82 vCPUs, 92.2 GiB RAM, 4 GPUs) and Node 2 (32 vCPUs, 42.8 GiB RAM, 1 GPU). The Kubernetes Default Scheduler cannot place all tasks on this configuration and requires an extra `n1-standard-16` node. For the 10% subset, PTK Grouped also yields a two-node cluster: Node 1 (92 vCPUs, 88.1 GiB RAM, 4 GPUs) and Node 2 (76 vCPUs, 109.6 GiB RAM, 4 GPUs). Kubernetes Default needs an extra `n1-standard-64` node with 4 GPUs to fit all Pods. PTK reserves a margin for Kubernetes overhead, which explains small gaps between requested and provisioned resources.

The default scheduler treats Pods independently and lacks whole-pipeline awareness, which scatters bandwidth-heavy neighbors and inflates cross-node transfers and cluster size. PTK's pre-deployment provisioning and placement co-locate heavy edges within Pods and shrink nodes after mapping, improving performance and reducing cost.

As summarized in Table III, PTK (Grouped) shortens the pipeline makespan by $6.43\times$ (5%) and $2.88\times$ (10%) relative to the Kubernetes Default Scheduler and reduces cost per run by $7.26\times$ and $4.28\times$, respectively. The gains stem from grouping bandwidth-heavy neighbors (`augment` and `preprocess`, `test_prepare` and `evaluate`), which reduces cross-node transfers and, together with right-sized provisioning, yields higher CPU/memory utilization. PTK achieves higher utilization due to right-sized provisioning and locality-aware mapping.

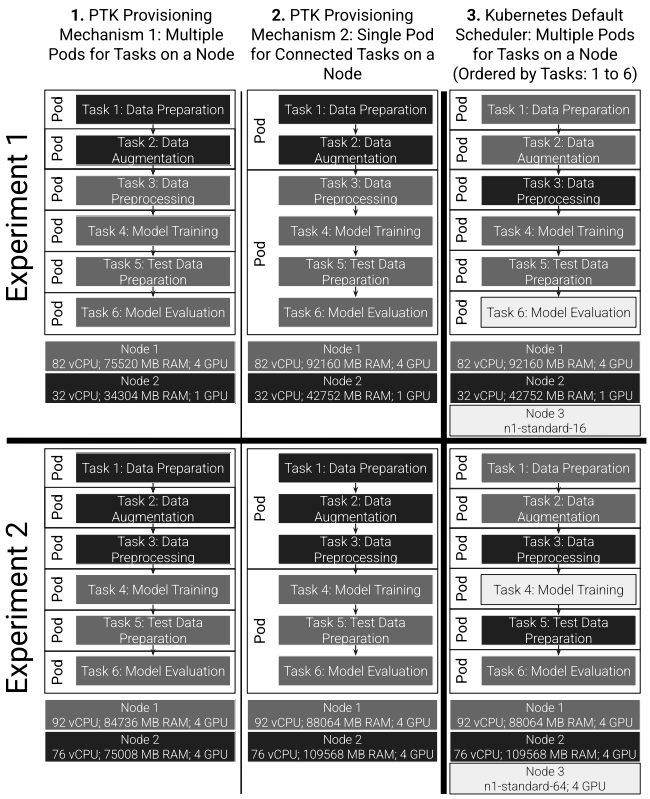

Fig. 3. Deployment configurations.

V. CONCLUSION

PTK combines lightweight annotations with application-aware provisioning to improve the price-performance of task-based Python on Kubernetes. On an ImageNet/ResNet50 pipeline, PTK co-locates bandwidth-heavy neighbors, right-sizes nodes, and delivers substantial runtime and cost reductions relative to the Kubernetes Default Scheduler.

REFERENCES

[1] Kubernetes.io, "Kubernetes: Open-Source Container Orchestration System," 2024, [Accessed 14-Sep-2024]. [Online]. Available: https://kubernetes.io/

[2] Google, LLC, "Tensorflow," 2025, [Accessed 28-Mar-2025]. [Online]. Available: https://www.tensorflow.org/

[3] Stanford Vision Lab, Stanford University, Princeton University, "Image Net: ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)," 2024, [Accessed 14-Sep-2024]. [Online]. Available: https://image-net.org/challenges/LSVRC/2012/2012-downloads.php

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, pp. 211–252, 2015.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[6] A. Nagiyev, E. Bajrovic, and S. Benkner, "Python to kubernetes: A programming and resource management framework for compute- and data-intensive applications," in *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2024, pp. 479–486.