Hybrid Reactive Autoscaling for Task-Based Pipelines on Kubernetes

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 hybrid autoscaling framework for pipeline-oriented, task-based Python applications on Kubernetes. PTK coordinates queue-length-driven horizontal scaling for CPU, memory, and GPU, together with reactive in-place vertical scaling of CPU and memory. The framework introduces source-code annotations, enabling users to define task-specific scaling constraints and automatically generate Kubernetes manifests. A periodic controller uses utilization and queue metrics to coordinate horizontal and vertical scaling, improving resource efficiency while maintaining pipeline performance. In a streaming machine learning (ML) inference pipeline, PTK sustains the target throughput while reducing hourly cost by 40.6%, CPU by 32.1%, and memory by 22.4%, and lowering the GPU count from 4 to 3, compared with an uncoordinated baseline that combines the Horizontal Pod Autoscaler (HPA) and the Vertical Pod Autoscaler (VPA). It also cuts peak cost by 23.6% compared with a queue-driven HPA baseline.**

***Index Terms*—autoscaling, Horizontal Pod Autoscaler, Kubernetes, resource management, source-code annotations, task-based programming, Vertical Pod Autoscaler**

I. Introduction

Over the past decade, hardware has become markedly more heterogeneous and cloud-centric, while software complexity has risen with the convergence of compute- and data-intensive applications. Managing these complexities effectively requires advanced autoscaling mechanisms. Autoscaling has become critical in modern cloud environments, especially for applications deployed on container-orchestration platforms such as Kubernetes [1]. Efficient resource management is essential for maintaining application responsiveness, optimizing resource utilization, and reducing operational costs. Autoscaling enables applications to dynamically adapt to workload fluctuations, preventing performance degradation and service interruptions under high load, while avoiding unnecessary resource allocation and cost wastage during low-demand periods.

Kubernetes provides two primary built-in autoscaling mechanisms: the HPA and the VPA. Both autoscalers operate at the granularity of Pods, which are the smallest deployable units within Kubernetes clusters. The HPA adjusts the number of Pod replicas based on short-term resource metrics, typically CPU utilization, to improve application throughput (i.e., the volume of data processed per unit of time) and manage immediate load spikes. Conversely, the VPA focuses on vertical scaling by recommending CPU and memory allocations within Pods, based on historical usage patterns and average resource demands observed over extended periods; by default, it applies changes by evicting and recreating Pods rather than updating them in place, which delays responsiveness to short-term bottlenecks. Cloud-managed Kubernetes services (e.g., Google Kubernetes Engine (GKE)) include these autoscalers, facilitating deployment and scaling.

Despite their effectiveness in certain scenarios, these autoscalers have limitations. First, the HPA and the VPA operate independently and consider scaling dimensions separately, lacking integration and coordination. Second, the VPA's reliance on historical data makes it insufficiently reactive, resulting in delayed responses to short-term performance bottlenecks. Third, both autoscalers focus on individual Pods rather than the multi-stage pipeline (i.e., the chain of tasks or functions deployed as multiple Kubernetes Pods and connected by inter-Pod queues) as a cohesive unit. Fourth, the built-in HPA scales replicas on CPU or memory utilization (GPU scale-out requires exposing a custom metric). The VPA, in turn, right-sizes only CPU and memory. However, neither autoscaler accounts for inter-Pod network communication overhead, which becomes critical in large, multi-node inference pipelines where stages run across multiple nodes and exchange high-volume data. Finally, deploying replicas to additional nodes without application-aware management can unintentionally increase network latency and degrade overall pipeline performance.

To address these limitations, we propose a runtime-adaptive autoscaling mechanism within the PTK framework [2]. The hybrid mechanism integrates reactive vertical and queue-driven horizontal scalings for pipeline-oriented, task-based applications on Kubernetes. Scalings are controlled by the user through source-code annotations defining task-specific autoscaling constraints taken into account by our Autoscaling Controller. Our approach leverages real-time, application-level metrics, including resource utilization (CPU, memory, GPU) and task queue lengths, to dynamically coordinate horizontal and vertical scalings. By proactively addressing resource bottlenecks and pipeline-level dependencies, PTK maintains throughput while improving resource and cost efficiency.

The remainder of this paper is structured as follows. Section II reviews related work. Section III provides an overview of PTK and describes the Python-based annotations for specifying resource requirements, autoscaling constraints, and deployment configurations. Section IV details the autoscaling mechanism. Section V presents an experimental evaluation of PTK's autoscaling mechanism applied to an ML pipeline. Sec-

tion VI summarizes the paper and discusses future directions.

II. Background and Related Work

Research on autoscaling for containerized systems has aimed at improving predictability and utilization and reducing cost. Prior analyses of the HPA have investigated metric choices and control behavior. For example, Casalicchio et al. [3] analyzed the Kubernetes HPA and proposed using absolute CPU performance metrics for resource management (e.g., CPU core units) instead of the default Kubernetes HPA approach, which relies on relative, percentage-based CPU utilization metrics. They demonstrated that using absolute metrics yielded more predictable and effective scaling behavior, particularly for CPU-intensive containerized workloads.

Vertical scaling and proactive rightsizing have also been studied. Shan et al. [4] proposed an adaptive anticipatory resource allocation scheme for workflow tasks to improve provisioning accuracy and reduce waste. A closely related line of work is the integration of vertical and horizontal scaling. Rzadca et al. [5] combined rightsizing with Kubernetes autoscaling in production clusters to improve resource management efficiency. Similarly, Vu et al. [6] proposed combining VPA and HPA with predictive, ML-based forecasting and burst detection to proactively handle demand spikes, aiming to maintain Quality of Service (QoS) and enhance resource utilization efficiency. However, these approaches do not model pipeline-level dependencies or inter-task communication overhead—factors that become prominent for data-intensive, task-based pipelines.

Beyond Kubernetes-native autoscalers, task-based programming models such as StarPU [7] and OpenMP tasking [8] deliver dynamic scheduling across heterogeneous resources in HPC settings but lack cloud-native support for autoscaling in containerized deployments. Related work has also explored high-level pipeline parallelism and pipeline patterns on many-core systems [9]–[11], but these do not address Kubernetes integration or queue-driven autoscaling.

Kubernetes Event-driven Autoscaling (KEDA) [12] complements HPA by exposing external, event-derived signals (queue length or stream lag) to drive activation and horizontal scale-out. KEDA supports scaling to and from zero replicas. Unlike PTK, which coordinates queue-aware horizontal actions with in-place vertical right-sizing and makes pipeline-level decisions across dependent tasks, KEDA focuses on trigger-driven horizontal scaling of individual workloads and does not orchestrate vertical resizing or pipeline-wide trade-offs.

Knative Pod Autoscaler (KPA) [13] provides request-driven autoscaling for stateless services. It scales based on either request concurrency or requests per second and uses separate stable and panic windows to smooth load and react to spikes. Unlike PTK, KPA does not perform vertical autoscaling.

III. Programming Framework

In this section, we provide an overview of the framework, discuss its task-based programming approach, and introduce source-code annotations for specifying resource requirements, autoscaling constraints, and deployment configurations.

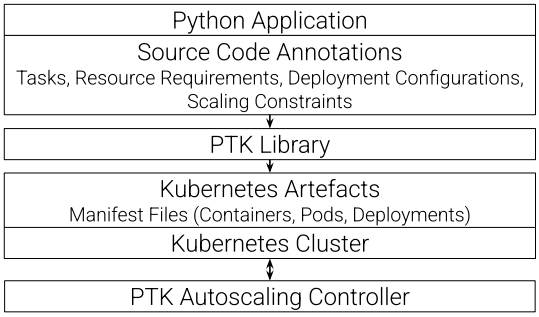

Fig. 1. Overview of the PTK framework.

A. Overview

The framework comprises a Python-based library and an autoscaling controller (Fig. 1). Users define application resources, autoscaling constraints, data-transfer hints, and task-to-Pod mappings by annotating Python functions. Based on these annotations, the framework transforms the application into Kubernetes manifest files containing objects (e.g., Deployments). Each Deployment manages a set of identical Pods, each of which contains one or more containers that encapsulate the application and its required dependencies, libraries, and runtime environment. PTK deploys manifests and co-deploys the autoscaling controller, which collects runtime metrics from the deployed workloads and orchestrates scaling decisions.

B. Task-Based Programming and Pipelines

To use PTK, each application must be decomposed into distinct tasks. Each task is represented as a Python function executed as a self-contained unit, including all necessary libraries and the runtime environment. Tasks communicate by sending and receiving data through runtime-managed queues: the output of one task is automatically enqueued and becomes the next task's input; users do not create or manage queues explicitly. Queues can reside on shared Network File System (NFS), local volumes, or in-memory stores, depending on performance and scalability requirements. PTK supports deployment on multi-node Kubernetes clusters, where tasks may run on different nodes. Communication between tasks is managed automatically by PTK via inter-Pod queues backed by Kubernetes storage (e.g., *PersistentVolumeClaims* (PVCs) or in-memory *emptyDir* volumes for co-located stages).

To demonstrate task-based programming and evaluate the effectiveness of our adaptive PTK autoscaling mechanism, we implement a synthetic real-time ML inference pipeline over the Galaxy10 DECaLS dataset [14] (Fig. 2). The dataset provides labeled galaxy image cutouts across ten morphological classes; the labels come from Galaxy Zoo [15], [16] and images from the DESI Legacy Imaging Surveys [17]. We use 256×256-px RGB cutouts as the input stream. The pipeline demonstrates key real-world challenges, including substantial variations in computational demands, stringent real-time constraints, GPU-intensive inference, and high-volume inter-task data transfers. The pipeline comprises the following sequential steps:

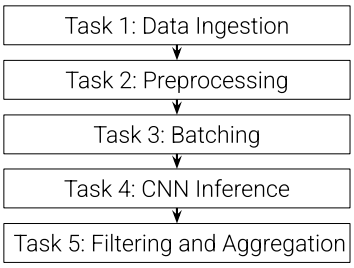

Fig. 2. ML inference pipeline.

Task 1 (Data Ingestion): This step reads Galaxy10 cutouts from shared storage (e.g., NFS), decodes them into tensors, and enqueues them for downstream processing. To sustain GPU utilization and emulate streaming workloads, images are grouped into batches that are published as single queue messages at a fixed cadence; in our experiments, we used batches of 50–300 images every 100 ms, yielding 500–3,000 images/s (≈210 MiB/s, assuming ∼145 KiB per PNG cutout).

Task 2 (Preprocessing): This step performs ML preprocessing (e.g., resizing/cropping, channel-wise normalization, photometric standardization) and prepares per-image metadata.

Task 3 (Batching): This step re-batches arriving batches into fixed-size batches of 128 images. Images are appended to a staging buffer as they arrive; a batch is dispatched immediately when it reaches 128, while any remainder is carried into the next batch. The stage coalesces queue messages into contiguous tensors and aligns the memory layout. Using a constant batch size improves kernel selection and occupancy, and amortizes I/O and launch overheads, leading to higher and more stable throughput in the subsequent inference step.

Task 4 (CNN Inference): This step runs a Convolutional Neural Network (CNN) based on the ResNet architecture [18] on GPUs to produce class-probability vectors over the ten Galaxy10 classes. To maximize throughput, the stage exploits batching and overlap of data transfers and computation. The output is a structured set of classification predictions, provided as probability vectors indicating the likelihood of each image belonging to one of the primary categories.

Task 5 (Filtering and Aggregation): This step collects and aggregates prediction results to enhance accuracy and reliability. Threshold-based filtering is applied to the CNN-generated probabilities, discarding predictions below predefined confidence levels. Duplicate or overlapping detections across different image segments are identified and removed to reduce redundancy. Final results are persisted for evaluation.

C. Application Annotations

PTK provides source-code annotations enabling users to define, for each application task, resource requirements, autoscaling constraints, deployment configurations, and inter-task data transfers directly in the application code prior to deployment (Listing 1 for the Galaxy10 pipeline).

1) Resource Requirements and Autoscaling Constraints: Within each `@task` annotation, users specify resource requirements and autoscaling constraints. CPU and memory are specified as Python tuples `(requests, limits)`. Resource *requests* define the minimum resources used for scheduling placement; the Kubernetes scheduler places Pods based on *requests*, while *limits* cap the maximum resources a container may use at runtime. These values serve as inputs to PTK's horizontal- and vertical-scaling logic. On Kubernetes v1.33 clusters with in-place Pod vertical scaling for CPU and memory enabled, PTK updates container *requests* and *limits* in place, i.e. without evicting and recreating Pods [19]. PTK computes the vertical-scaling actions itself and applies the updates directly via the Kubernetes API. To prevent drastic resource fluctuations and ensure stability, PTK introduces `cpu_thresholds` and `memory_thresholds` annotations, each specified as a pair of tuples for *requests* and *limits* – `(min, max)`. These thresholds bound the allowable resource adjustments, enabling the autoscaler to modify initial *requests* and *limits* within user-defined ranges. Listing 1 shows these thresholds for the `preprocess` task.

The framework provides horizontal queue-based autoscaling through `pending_res=(min, max)`, which specifies lower and upper thresholds for the input queue of a task. In a linear pipeline this input queue is identical to the upstream stage's output, but the scaling action is applied to the consuming task. When the queue length exceeds `max`, PTK scales up the task to mitigate throughput bottlenecks. If the queue length falls below `min`, it scales down the task to optimize resource utilization. The annotation supports thresholds based on the number of queued results or the total data size. For example, specifying `pending_res=('50MiB', '500MiB')` for the Preprocessing step instructs PTK to scale down when the queue is fewer than 50 MiB, and scale up whenever the total queued data awaiting processing exceeds 500 MiB.

Users define horizontal autoscaling constraints using `replicas=(min, max)`, the allowed range of Pod replicas. The PTK Autoscaling Controller scales replicas within this range when the task's input queue length crosses the configured `pending_res` bounds (with a CPU- and memory-based downscale guard). Because Kubernetes cannot resize GPU resources in place, PTK fixes GPU allocation per replica. Thus, increasing GPU capacity for a task is achieved via horizontal scaling (i.e., increasing the number of Pod replicas).

The `utilization_bounds` enables users to define lower and upper bounds for CPU and memory utilization that trigger reactive vertical scaling. Lower values (e.g., 0.3 or 30%) initiate resource downscaling earlier, improving efficiency when workloads consistently underutilize allocated resources. Conversely, smaller upper values (e.g., 70%) initiate resource upscaling earlier, ensuring additional resource headroom beneficial for workloads with highly variable demands. Higher upper values (e.g., 90%) yield more conservative resource allocation, maximizing resource utilization and minimizing overhead—suitable for stable workloads. By default, PTK uses 50% and 90%, and evaluates adjustments every 60 s.

Each task may optionally set `gamma_up` and `gamma_down`, the scale-up and scale-down step fractions

that bound how many replicas change per decision, and `downscale_guard` $\in (0, 1]$, a utilization-ratio threshold applied to CPU and memory (and GPU when available) below which horizontal downscale is permitted. Unless specified, PTK uses conservative defaults: `gamma_up` is 0.5, `gamma_down` is 0.25, and `downscale_guard` is 0.5.

2) Deployment Configurations: PTK provides annotations to define the mapping of tasks to Pods and containers [2]. Multiple tasks can be grouped into a single container, and multiple containers can be deployed within a single Pod. Users configure these mappings with `@pod`, which specifies task-to-Pod allocation, and `@container`, which specifies the task-to-container mapping.

Fine-tuning the organization of tasks into Pods and containers influences scalability and performance, revealing an important trade-off. Packing multiple containers into one Pod enhances data-transfer bandwidth by enabling in-memory communication; however, this approach restricts the deployment of those tasks across multiple nodes, thereby limiting horizontal scalability. Conversely, distributing tasks across separate Pods improves horizontal scalability but may introduce additional inter-node network overhead, potentially reducing data-transfer efficiency. If users apply the `@task` annotation without further specification, PTK creates a separate Pod and container for each task by default.

3) Data Transfer: Beyond Kubernetes' standard capabilities, PTK handles data transfers between tasks, which can significantly affect overall pipeline performance. Users specify approximate data-transfer sizes via `input_data` and `output_data` clauses for each annotation. PTK uses these hints to provision appropriate Kubernetes-based storage resources, such as local `PersistentVolumes` or NFS-backed storage based on HDD and SSD managed through Kubernetes' PVCs, for tasks deployed on separate Pods. For tasks residing within the same Pod, PTK employs an in-memory data-transfer mechanism using Kubernetes' temporary shared volume *emptyDir* with `medium: "Memory"`, facilitating high-performance communication. In our pipeline, NFS-backed PVCs are used for inter-Pod queues.

Task connectivity follows standard Python conventions: function arguments represent inputs and return values represent outputs. By analyzing these annotations and task arguments, PTK ensures that dependent tasks residing in different Pods and containers transfer data correctly. For automatic serialization and deserialization of transferred data, PTK leverages Python's built-in `json` module for standard Python types and the external `NumPy` library for array data, enabling seamless data exchange within the pipeline.

IV. Autoscaling Mechanism

In this section, we introduce the adaptive autoscaling mechanism integrated into PTK and describe its core components (shown in Fig. 3). We address how PTK supports reactive vertical scaling and coordinates vertical and horizontal scaling actions to optimize pipeline-level autoscaling decisions.

```
from PTK import task

@task(name='ingest', cpu=(4, None), memory=('
    10GiB', None), input_data={'size':'6GiB'},
     pending_res=('50MiB', '500MiB'),
    output_data={'size':'6GiB', 'type':'nfs'})
def ingest():
    # Task 1: Ingest raw data from source
    return ingest_result

@task(name='preprocess', cpu=(16, 48), memory
    =('10GiB', '20GiB'), replicas=(1, 3),
    cpu_thresholds=((8, 32), (16, 64)),
    memory_thresholds=(('8GiB', '32GiB'), ('16
    GiB', '64GiB')), pending_res=('50MiB', '
    500MiB'), output_data={'size':'6GiB', '
    type':'nfs'}, gamma_up=0.5, gamma_down
    =0.25, downscale_guard=0.5)
def preprocess(ingest_result):
    # Task 2: Preprocess data
    return preprocess_result

@task(name='batch', cpu=(4, None), memory=('8
    GiB', None), utilization_bounds=(0.3, 0.8)
    , pending_res=('50MiB', '500MiB'),
    output_data={'size':'6GiB', 'type':'nfs'})
def batch(preprocess_result):
    # Task 3: Reorganize batches
    return batch_result

@task(name='run_inference', cpu=(2, None),
    gpu=1, memory=('6GiB', None), pending_res
    =('50MiB', '500MiB'), output_data={'size':
    '8GiB', 'type':'nfs'})
def run_inference(batch_result):
    # Task 4: CNN-based classification
    return run_inference_result

@task(name='filter_aggregate', cpu=(2, None),
     memory=('4GiB', None), pending_res=('50
    MiB', '500MiB'), output_data={'size':'8GiB
    ', 'type':'nfs'})
def filter_aggregate(run_inference_result):
    # Task 5: Filter results
    return filter_result

def main():
    ingest_res = ingest()
    prep_res = preprocess(ingest_res)
    batch_res = batch(prep_res)
    infer_res = run_inference(batch_res)
    filter_res = filter_aggregate(infer_res)
```

Listing 1. Galaxy10 DECaLS inference pipeline

A. PTK Autoscaling Components

The proposed PTK autoscaling mechanism comprises two core components that operate collaboratively to enable dynamic pipeline-level autoscaling within Kubernetes clusters.

Metrics Agent: A lightweight Metrics Agent is implemented as a Kubernetes DaemonSet configured to run one Pod per node and continuously collects runtime utilization metrics (CPU, memory, GPU) and queue metrics (per-task queue

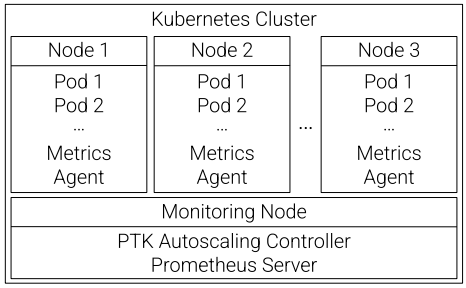

Fig. 3. Core components of our autoscaling framework.

length and wait time). CPU/memory come from Kubernetes; GPU metrics are exported via the NVIDIA Data Center GPU Manager (DCGM) when GPUs are present. Queue metrics are emitted by the PTK library at the task ingress/egress points to avoid log scraping. Metrics are exported in real time to the Prometheus server [20] at 5 s intervals, enabling rapid aggregation and analysis of metrics to support timely pipeline-level autoscaling decisions.

PTK Autoscaling Controller: Implemented using Kubebuilder [21], the controller orchestrates autoscaling decisions based on time-series metrics provided by Prometheus via its HTTP API, which are systematically collected by Metrics Agents. At fixed decision intervals (default: 60 s), the controller computes coordinated horizontal (replica counts) and vertical (CPU/memory *requests* and *limits*) actions per task over a stabilization window (default: 300 s). To prevent oscillations, PTK enforces one action per decision interval and records the timestamp of the last change. If metrics are stale or unavailable for a task, scaling for that task is temporarily frozen until fresh samples are observed.

Leveraging Kubernetes v1.33 in-place Pod vertical scaling for CPU and memory, the controller updates container *requests* and *limits* without restarting Pods [19]. We optionally use the upstream VPA recommender for target hints but do not rely on VPA's eviction-based update path. For horizontal actions, the controller writes the target replica count via the standard scale subresource of the owning workload. All changes are enacted through the Kubernetes API.

The controller, Prometheus Server and Metrics Agent are deployed alongside each application, providing real-time monitoring and continuous optimization of autoscaling decisions.

B. Horizontal and Vertical Scaling

By default, Kubernetes' HPA and VPA operate independently, which can lead to conflicting or inefficient resource-allocation decisions. To address this limitation, we present a dynamic autoscaling mechanism integrated into PTK that seamlessly combines reactive vertical and horizontal scaling. This unified approach is executed by the PTK Autoscaling Controller at each decision interval, applying HPA or VPA holistically as appropriate to proactively adjust per-Pod resources and replica counts and to optimize task performance, resource efficiency, and pipeline responsiveness. The proposed scaling strategy consists of two complementary phases:

Horizontal Scaling Phase: The PTK framework dynamically adjusts the replica count r_i^{r} for each task t_i based on the current queue length q_i and resource utilization u_i and a per-task stabilization window extending Kubernetes' standard HPA. Algorithm 1 illustrates scaling logic only for CPU, but identical logic is applied to memory and GPU, currently assuming homogeneous GPU types across the cluster.

The `pending_res` annotation defines the input-queue bounds $q_i^{\min}$ and $q_i^{\max}$ for task t_i; the queue length q_i is measured either in items or in aggregate bytes, as specified by the annotation. Horizontal scaling is triggered when the queue length crosses its predefined bounds, subject to the configured replica *limits*. Horizontal scaling is considered only when q_i crosses these bounds and a full stabilization window has elapsed since the last scaling change for t_i; all actions respect the replica *limits* $[r_i^{\mathrm{r,min}}, r_i^{\mathrm{r,max}}]$.

When $q_i \geq q_i^{\max}$ and the task has not reached its maximum replicas ($r_i^{\mathrm{r}} < r_i^{\mathrm{r,max}}$), PTK increases r_i^{r} by a bounded step proportional to the current replica count, $\max\big(1,\ \lceil \gamma_{up} \cdot r_i^{\mathrm{r}} \rceil\big)$, where γ_{up} determines how many replicas can be added per decision and guarantees that at least one is added when γ_{up} is 0. It then limits the new replica count $r_i^{\mathrm{r}\prime}$ to $r_i^{\mathrm{r,max}}$. Conversely, when $q_i \leq q_i^{\min}$ and the task remains above its minimum replicas ($r_i^{\mathrm{r}} > r_i^{\mathrm{r,min}}$), PTK decreases r_i^{r} by a bounded step controlled by γ_{down} only if the per-replica CPU, GPU or memory utilization ratio (usage u_i relative to *request* r_i) is below the guard threshold h; otherwise no downscale is applied. Both q_i and u_i are averaged over the stabilization window, and the controller records the time of each change to suppress further actions within the same window.

At each decision interval, the controller runs the same horizontal scaling three times, once each for CPU, memory, and GPU, so it gets three replica proposals for the task. They look at the task's input-queue pressure but use the resource-specific utilization to decide whether scale-in is safe. The controller merges these proposals conservatively: on scale-out, it picks the largest of the three; on scale-in, it reduces replicas only if all three passes agree that it is safe to shrink; otherwise, it keeps the current replica count. This prevents removing capacity while any resource is still a bottleneck.

Vertical Scaling Phase: The PTK framework dynamically adjusts CPU and memory resource *requests* r_i and *limits* l_i of existing Pods for each task t_i based on the mean per-replica usage u_i measured over the stabilization window (Algorithm 2). While the algorithm illustrates CPU scaling, the logic for memory scaling is identical. PTK uses lower and upper utilization bound $h_i^{\min}$ and $h_i^{\max}$, with default values of 50% and 90%. Vertical scaling is considered when either (i) the queue length q_i is within its bounds $[q_i^{\min}, q_i^{\max}]$ instantiated by the `pending_res` annotation, or (ii) the task has reached its maximum replica count ($r_i^{\mathrm{r}} = r_i^{\mathrm{r,max}}$).

When utilization relative to the current *request* exceeds the upper bound $h_i^{\max}$, PTK scales the *request* up so that the predicted utilization falls back near the upper bound, rounding to scheduler units q (by default, 0.1 cores and 128 MiB), ensuring resource values remain within the defined maximum

Algorithm 1: PTK Horizontal Scaling (CPU)

Input:
$T = \{t_1, \ldots, t_t\}$: set of t tasks; each task t_i is defined by:
- r_i^{r}: current replicas with bounds $[r_i^{\mathrm{r,min}}, r_i^{\mathrm{r,max}}]$;
- q_i: current queue length with bounds $[q_i^{\min}, q_i^{\max}]$;
- r_i: per-replica CPU request ($r_i > 0$);
- u_i: per-replica CPU usage, averaged over the stabilization window;
- γ_{up}, γ_{down}: scale-up / scale-down step fractions;
- h: downscale guard (utilization-ratio threshold).

Output:
$T' = \{t'_1, \ldots, t'_t\}$: updated tasks with adjusted replicas ${r_i^{\mathrm{r}}}'$.

foreach *task* $t_i \in T$ **do**
  if $q_i \geq q_i^{\max}$ ***and*** $r_i^{\mathrm{r}} < r_i^{\mathrm{r,max}}$ **then**
    ${r_i^{\mathrm{r}}}' \leftarrow \min\big(r_i^{\mathrm{r}} + \max\big(1,\ \lceil \gamma_{up} \cdot r_i^{\mathrm{r}} \rceil\big),\ r_i^{\mathrm{r,max}}\big)$;
  else if $q_i \leq q_i^{\min}$ ***and*** $r_i^{\mathrm{r}} > r_i^{\mathrm{r,min}}$ ***and*** $\dfrac{u_i}{r_i} < h$ **then**
    ${r_i^{\mathrm{r}}}' \leftarrow \max\big(r_i^{\mathrm{r}} - \max\big(1,\ \lceil \gamma_{down} \cdot r_i^{\mathrm{r}} \rceil\big),\ r_i^{\mathrm{r,min}}\big)$;
  else ${r_i^{\mathrm{r}}}' \leftarrow r_i^{\mathrm{r}}$;
return T'

Algorithm 2: PTK Vertical Scaling (CPU)

Input:
$T = \{t_1, \ldots, t_t\}$: set of t tasks; for each t_i:
- r_i^{r}: current replicas with bounds $[r_i^{\mathrm{r,min}}, r_i^{\mathrm{r,max}}]$;
- q_i: current queue length with bounds $[q_i^{\min}, q_i^{\max}]$;
- r_i: per-replica CPU request with bounds $[r_i^{\min}, r_i^{\max}]$;
- l_i: per-replica CPU limit with bounds $[l_i^{\min}, l_i^{\max}]$;
- u_i: per-replica CPU usage, averaged over the stabilization window;
- $h_i^{\min}$, $h_i^{\max}$: CPU utilization bounds;
- q_{cpu}: CPU scheduler quantum (e.g., 0.1 cores).

Output:
$T' = \{t'_1, \ldots, t'_t\}$: updated tasks with adjusted CPU requests and limits (r'_i, l'_i).

foreach *task* $t_i \in T$ **do**
  if $q_i^{\min} \leq q_i \leq q_i^{\max}$ ***or*** $r_i^{\mathrm{r}} = r_i^{\mathrm{r,max}}$ **then**
    if $\dfrac{u_i}{r_i} \geq h_i^{\max}$ **then**
      $r'_i \leftarrow \min\left(\left\lceil \frac{u_i}{h_i^{\max}} \right\rceil_{q_{\text{cpu}}},\ r_i^{\max}\right)$;
    else if $\dfrac{u_i}{r_i} \leq h_i^{\min}$ **then**
      $r'_i \leftarrow \max\left(\left\lfloor \frac{u_i}{h_i^{\min}} \right\rfloor_{q_{\text{cpu}}},\ r_i^{\min}\right)$;
    else $r'_i \leftarrow r_i$;
    if $r'_i \neq r_i$ **then**
      $l'_i \leftarrow \min\left(l_i^{\max},\ \max\left(\left\lceil \frac{r'_i \cdot l_i}{r_i} \right\rceil_{q_{\text{cpu}}},\ r'_i,\ l_i^{\min}\right)\right)$;
    else $l'_i \leftarrow l_i$;
return T'

thresholds $r_i^{\max}$. The corresponding *limit* is increased consistently (preserving the *request* and *limit* ratio when possible), then clamped to $[l_i^{\min}, l_i^{\max}]$ from `cpu_thresholds` and `memory_thresholds` while ensuring $l'_i \geq r'_i$.

Conversely, when utilization drops below the lower bound, it scales the *request* down so that the predicted utilization returns to the band, rounding down to scheduler units and never going below $r_i^{\min}$. The *limit* is reduced in step with the *request* and clamped to $[l_i^{\min}, l_i^{\max}]$, preserving $l'_i \geq r'_i$. If utilization lies within the band, both *requests* and *limits* are left unchanged. Due to current Kubernetes constraints, PTK performs vertical scaling only for CPU and memory; GPUs cannot be resized in place and are addressed via horizontal scaling.

Taken together, horizontal scaling and vertical scaling enable the framework to maintain throughput and responsiveness while improving efficiency and stability. To avoid vertical and horizontal interference, PTK (i) bases horizontal decisions on external queue-length signals rather than CPU-percentage of *request*, (ii) performs at most one action (replica or *request* and *limit* change) per decision interval and during which no further scaling is applied to the pipeline. We configure the stabilization windows and rate *limits* to suppress oscillations.

V. Experiments

To empirically evaluate the effectiveness of the dynamic autoscaling mechanism, we constructed scenarios for the Python-based ML inference pipeline described in Section III-B.

A. Overview

We construct a streaming image-classification inference pipeline from the Galaxy10 dataset (256×256-px cutouts; each image ≈145 KiB) as the input stream (Section III-B). Using the same pipeline, to vary compute intensity at fixed network throughput, we also consider GalaxiesML (127×127-px cutouts; each image ≈35 KiB) [22]. In both cases, we form synthetic batches to produce controlled input throughput.

We evaluate our proposed PTK autoscaling strategy against two Kubernetes baselines, where all strategies use identical min/max replica bounds per task:

(i) *HPA+VPA (uncoupled)*: HPA with an 80% CPU target (300 s stabilization) and VPA in Auto mode with evicting updates; only CPU/memory metrics are used (no queue or GPU signals); in-place vertical resizing is applied by PTK based on VPA recommendations.

(ii) *KEDA*: HPA driven by an external queue-length metric (5 min average) implemented via KEDA; scale-to-zero disabled; 300 s for both stabilization and cooldown periods; no vertical scaling. Polling interval is 30 s. We treat KEDA as a baseline because it uses the same queue-length signal as PTK for horizontal scaling. Replica bounds are identical to PTK. The queue threshold equals the `pending_res` bounds.

(iii) *PTK*: our controller from Section IV, combining queue-aware horizontal scaling with reactive in-place vertical scaling (CPU/memory) using a utilization band of $[0.5, 0.9]$ and a 300 s stabilization window; decisions every 60 s.

Experiments run on Google Cloud Platform (GCP) on a homogeneous node pool with three `n1-standard-64` nodes in `us-central1`; each node has four NVIDIA Tesla T4 GPUs (16 GiB). A dedicated CPU-only node (`n1-standard-4`) hosts the PTK Autoscaling Controller and Prometheus (pinned via a Kubernetes node selector). The CNN inference task uses CUDA 12.4 on GPU; other tasks are CPU-only. Inter-task queues use NFS-backed PVCs (`ReadWriteMany`) so that replicas across nodes consume shared data consistently. We

TABLE I
EXPERIMENT 1: DATA-INTENSIVE SCENARIOS

Autoscaling Strategy	Scenario	Throughput		Provisioned resources			Cost ($/h)
		img/s	MiB/s	CPU	Memory	GPU	
HPA+VPA	1	500	70	34	50	1	1.72
	2	1,500	210	64	112	2	3.36
	3	3,000	420	112	192	4	6.04
KEDA	1	500	70	34	52	1	1.73
	2	1,500	210	56	100	2	3.04
	3	3,000	420	96	170	4	4.70
PTK	1	500	70	32	53	1	1.67
	2	1,500	210	48	87	2	2.71
	3	3,000	420	76	149	3	3.59

Note: CPU values are vCPU cores; Memory values are in GiB.

discard the first 10 min as warm-up and average metrics over the subsequent 60 min.

We model cost using the GCP Pricing Calculator [23]. For each task, we extract the steady-state provisioned resources over the reporting window—i.e., the average replica count multiplied by the per-replica *requests* with the number of GPUs. In `us-central1` and at on-demand rates, we price the smallest single VM configuration that can host that task footprint: for CPU/memory we select a custom `n1` machine with the required vCPUs and GiB (rounded up), and when GPUs are present we attach the minimal number of T4 devices needed for that task; this yields a per-task $/h. We sum tasks to obtain the modeled pipeline cost for each strategy. This cost reflects the resources used by the pipeline, not the cluster's bill, and is reported consistently across strategies with identical task-to-Pod mappings, images, and hardware. We held the cluster hardware, container images, and task-to-Pod mappings fixed across all experiments; only the input throughput and autoscaling policy varied.

B. Evaluation Scenarios

To evaluate and compare the performance of different autoscaling approaches, we conducted two distinct experimental scenarios (compute-intensive and data-intensive) designed to simulate realistic workload variations.

Experiment 1. Data-Intensity Sweep: To assess horizontal-scaling responsiveness under varying input pressure, we adjusted the input throughput by modifying the number of concurrent images per second, while keeping the per-image size (Galaxy10 cutouts, ≈145 KiB each) fixed, closely reflecting realistic operational conditions. We considered three scenarios to evaluate scalability and resource efficiency (see Table I):

1) *Moderate Load (baseline)*: 500 images/s (≈70 MiB/s).
2) *High Load*: 1,500 images/s (3× baseline; ≈210 MiB/s).
3) *Peak Load*: 3,000 images/s (6× baseline; ≈420 MiB/s).

All images differ in content but remain identical in size, ensuring consistent assessment criteria across all scenarios.

As the input throughput scales from 1× (baseline) to 3× and 6×, PTK sustains the target throughput with fewer resources than uncoordinated Kubernetes autoscaling. At moderate load, PTK reduces CPU cores by 5.9% (34 vs. 32) while allocating

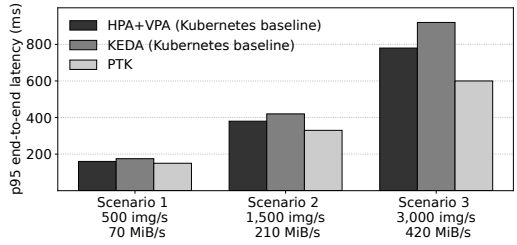

Fig. 4. p95 end-to-end latency by load scenario for the Galaxy10 dataset.

6.0% more memory (50 vs. 53) with the same GPU count, lowering cost by 3.0%. As PTK requires an additional node for Autoscaling Controller and Prometheus Server, it reduces differences in resource utilization and cost for scenarios with smaller data volumes. However, at high load, PTK shows better results, decreasing the number of CPU and memory by 25.0% and 22.3%, while keeping the GPU count the same, cutting cost by 19.4%. At peak load, PTK requires three GPUs instead of four and further reduces CPU and memory by 32.1% and 22.4%, yielding a 40.6% lower cost. These gains arise from coordinating HPA and VPA around queue-length bounds and stabilized utilization, avoiding the overprovisioning observed with independent autoscalers. Relative to KEDA, PTK achieves a smaller footprint at all loads by right-sizing CPU/memory in place and scaling replicas only when queues breach bounds. PTK provisions 20.8% less CPU and 12.4% less memory at peak load and requires 3 GPUs at peak, decreasing modeled cost by 23.6% compared with KEDA.

We measure per-image end-to-end latency as the time from ingest enqueue to result persistence at the sink, and report 95th percentile (p95) over a 60-min steady-state window following a 10-min warm-up (Fig. 4). The figure plots p95 latency for three data-intensive load scenarios; the y-axis shows latency (ms) for the two Kubernetes baselines (HPA+VPA and KEDA) and PTK, and the x-axis lists the three input-rate scenarios. Across all loads, PTK yields the lowest latency, improving over HPA+VPA by 6–23% and over KEDA by 14–35%.

Experiment 2. Compute Intensity at Fixed Throughput: To exercise vertical scaling while holding network pressure constant, we ran two workloads at a fixed input workload of ≈210 MiB/s, matching the high-throughput setting in Experiment 1. The baseline workload uses Galaxy10 cutouts, which yields ≈1,500 images/s (256×256 -px) at this data throughput (Scenario 1). The compute-intensive workload uses GalaxiesML cutouts, which yields ≈6,144 images/s (127×127 -px) at the same 210 MiB/s (Scenario 2).

However, due to the differences in image resolutions, the number of images processed per second varies significantly, thus changing the computational demands per second. The per-inference compute differs, but because the 127×127 stream yields ≈4× more images/s, total compute/s is higher, which primarily engages VPA. This design isolates and evaluates the effectiveness of PTK's vertical autoscaling capabilities under changing computational loads. In both experimental setups, we measured and compared pipeline resource utilization and

TABLE II
EXPERIMENT 2: COMPUTE INTENSITY AT FIXED THROUGHPUT

Autoscaling Strategy	Scenario	Throughput		Provisioned resources			Cost ($/h)
		img/s	MiB/s	CPU	Memory	GPU	
HPA+VPA	1	1,500	210	64	112	2	3.32
	2	6,144	210	72	128	2	3.66
KEDA	1	1,500	210	56	100	2	3.04
	2	6,144	210	62	108	2	3.24
PTK	1	1,500	210	48	87	2	2.68
	2	6,144	210	54	91	2	2.90

Note: CPU values are vCPU cores; Memory values are in GiB.

overall cost efficiency. We report the resulting steady-state resource footprints and costs in Table II.

Scenario 1 reproduces the high-throughput setting from Experiment 1. PTK sustains the target rate with the same GPU count while right-sizing CPU and memory, lowering modeled cost by 19.3% relative to HPA+VPA and 11.8% relative to KEDA. Scenario 2, with substantially more images/s at the same byte throughput, again uses 2 GPUs across all three strategies. HPA+VPA provisions 72 vCPUs and 128 GiB and KEDA 62 vCPUs and 108 GiB, whereas PTK right-sizes to 54 vCPUs and 91 GiB: 25.0% fewer CPU, 28.9% less memory, and 20.8% lower cost relative to HPA+VPA; and 12.9% fewer CPU, 15.7% less memory, and 10.5% lower cost relative to KEDA. Because queues remain within bounds, horizontal actions are infrequent; gains primarily come from in-place vertical scaling that keeps utilization within the target band.

VI. CONCLUSION

This paper presents PTK, a runtime-adaptive autoscaling orchestrator for deploying and scaling task-based pipelines on Kubernetes clusters. PTK provides Python source-code annotations that let users specify resource requirements, deployment configurations, data transfers, and autoscaling constraints prior to deployment. We presented a coordinated autoscaling mechanism that integrates reactive, queue-length-driven horizontal scaling and in-place vertical scaling.

We implemented a prototype and evaluated it on a streaming ML inference pipeline. It outperforms uncoordinated HPA+VPA and KEDA baselines, sustaining target throughput while reducing hourly cost by up to 41%, CPU by up to 32%, and memory by up to 29%, and lowering the GPU count in the peak-load scenario. PTK further reduces p95 end-to-end latency by approximately 6–23% compared with HPA+VPA and 14–35% compared with KEDA across scenarios.

Future research directions include enhancing PTK by introducing more advanced annotations (e.g., SLO- or latency-aware constraints), integration with cluster autoscaling mechanisms and heterogeneous node pools, especially those with advanced GPU configurations. Extending integration to multiple cloud providers beyond GKE will further increase the versatility and practical applicability of PTK.

REFERENCES

[1] Kubernetes.io, "Kubernetes: Open-Source Container Orchestration System." https://kubernetes.io/, 2024. [Online; accessed 10-May-2025].

[2] 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)*, pp. 479–486, IEEE, 2024.

[3] E. Casalicchio, "A study on performance measures for auto-scaling cpu-intensive containerized applications," *Cluster Computing*, vol. 22, no. 3, pp. 995–1006, 2019.

[4] C. Shan, C. Wu, Y. Xia, Z. Guo, D. Liu, and J. Zhang, "Adaptive resource allocation for workflow containerization on kubernetes," *Journal of Systems Engineering and Electronics*, vol. 34, no. 3, pp. 723–743, 2023.

[5] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[6] D.-D. Vu, M.-N. Tran, and Y. Kim, "Predictive hybrid autoscaling for containerized applications," *IEEE Access*, vol. 10, pp. 109768–109778, 2022.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, the Netherlands, August 25-28, 2009. Proceedings 15*, pp. 863–874, Springer, 2009.

[8] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[9] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault, "High-level support for pipeline parallelism on many-core architectures," in *European Conference on Parallel Processing*, pp. 614–625, Springer, 2012.

[10] E. Bajrovic and S. Benkner, "Automatic performance tuning of pipeline patterns for heterogeneous parallel architectures," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, (Las Vegas, Nevada, USA), July 21–24 2014.

[11] E. Bajrovic, S. Benkner, and J. Dokulil, "Pipeline patterns on top of task-based runtimes," in *International Conference on Parallel and Distributed Computing: Applications and Technologies*, pp. 100–110, Springer, 2018.

[12] The KEDA Authors, "Keda: Kubernetes event-driven autoscaling." https://keda.sh/, 2025. [Online; accessed 23-Sep-2025].

[13] "Knative autoscaling." https://knative.dev/docs/serving/autoscaling/, 2025. [Online; accessed 10-Aug-2025].

[14] H. W. Leung and astroNN contributors, "Galaxy10 decals dataset." *astroNN Documentation*, 2025. [Online; accessed 10-Sep-2025].

[15] C. J. Lintott, K. Schawinski, A. Slosar, K. Land, S. Bamford, D. Thomas, M. J. Raddick, R. C. Nichol, A. Szalay, D. Andreescu, *et al.*, "Galaxy zoo: morphologies derived from visual inspection of galaxies from the sloan digital sky survey," *Monthly Notices of the Royal Astronomical Society*, vol. 389, no. 3, pp. 1179–1189, 2008.

[16] M. Walmsley, C. Lintott, T. Géron, S. Kruk, C. Krawczyk, K. W. Willett, S. Bamford, L. S. Kelvin, L. Fortson, Y. Gal, *et al.*, "Galaxy zoo decals: Detailed visual morphology measurements from volunteers and deep learning for 314 000 galaxies," *Monthly Notices of the Royal Astronomical Society*, vol. 509, no. 3, pp. 3966–3988, 2022.

[17] A. Dey, D. J. Schlegel, D. Lang, R. Blum, K. Burleigh, X. Fan, J. R. Findlay, D. Finkbeiner, D. Herrera, S. Juneau, *et al.*, "Overview of the desi legacy imaging surveys," *The Astronomical Journal*, vol. 157, no. 5, p. 168, 2019.

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

[19] "Resize cpu and memory resources assigned to containers." Kubernetes Documentation, 2025. Feature state: Kubernetes v1.33 [beta]. [Online; accessed 10-Sep-2025].

[20] The Prometheus Authors, "Prometheus: Monitoring system & time series database." https://prometheus.io/, 2016. [Online; accessed 17-Sep-2025].

[21] Kubebuilder Google Group, "Kubebuilder: Kubernetes API Builder." https://book.kubebuilder.io/, 2024. [Online; accessed 10-May-2025].

[22] T. Do, B. Boscoe, E. Jones, Y. Q. Li, and K. Alfaro, "Galaxiesml: a dataset of galaxy images, photometry, redshifts, and structural parameters for machine learning," *arXiv preprint arXiv:2410.00271*, 2024.

[23] Google Cloud, "Google cloud pricing calculator." https://cloud.google.com/products/calculator, 2025. [Online; accessed 10-Sep-2025].