Python to Kubernetes: A Programming and Resource Management Framework for Compute- and Data-intensive Applications

Andrey Nagiyev
Faculty of Computer Science
University of Vienna
Vienna, Austria
andrey.nagiyev@univie.ac.at

Enes Bajrovic
Faculty of Computer Science
University of Vienna
Vienna, Austria
enes.bajrovic@univie.ac.at

Siegfried Benkner
Faculty of Computer Science
University of Vienna
Vienna, Austria
siegfried.benkner@univie.ac.at

***Abstract*—In this paper, we introduce the Python to Kubernetes (PTK) framework, a high-level Python-based programming framework for deploying Python applications on top of Kubernetes clusters. PTK supports a task-based programming approach with extensions for specifying resource requirements and performance constraints. A major goal of PTK is to provide users with high-level control for deploying compute- and data-intensive applications on different types and configurations of heterogeneous clusters, while ensuring performance and/or cost constraints.**

***Index Terms*—compute-intensive applications, data-intensive applications, task-based programming, resource management, clustering, containerization**

I. Introduction

Over the past decades, we have seen a sharp increase in the complexity of computer architectures, mainly caused by parallelism and heterogeneity. Moreover, we have observed an ever-growing complexity of applications due to the need of combining compute- and data-intensive tasks. As a result of physical limits in the design of processor architectures, faster computing systems can now only be achieved by increasing the degree of parallelism and utilizing heterogeneous systems with specialized processing units, such as GPUs or tensor processing units, which can accelerate certain types of computations. In addition, we witness the emergence of a computing continuum that encompasses systems at vastly different scales, ranging from mobile devices to servers, supercomputers, and cloud data centers [1], [2].

Alongside the dramatic developments in the performance of computing systems, we observe a convergence of compute-intensive and data-intensive applications. Examples include scientific simulations tightly coupled with data analytics [3]–[5], or machine learning (ML) pipelines that process massive amounts of data to train large-scale neural networks.

The development of compute- and data-intensive applications for heterogeneous parallel systems is associated with many challenges. Besides the challenge of parallel programming, users also need to deal with intricate resource management tasks, performance optimization, and portability issues. Given the large spectrum of computer architectures, portability of applications is a key issue, since ideally, an application should be able to run (efficiently) across a range of different architectures and systems. Users want to be able to develop their applications on a small local system, while being able to deploy the final application on large clusters or in the cloud. When running an application in the cloud, users would like to control the trade-off between performance/execution time and cost. Additionally, users would like to seamlessly utilize special hardware, e.g., GPUs, when they are available in a system to accelerate certain performance-critical tasks in their applications. Finally, users want to get rid of low-level infrastructure management tasks as far as possible, which are, however, often required when running applications on cloud-provisioned or virtualized resources.

In this paper, we address the challenges associated with the development of complex applications, and their execution across a diverse landscape of architectures by proposing the Python-to-Kubernetes (PTK) framework. PTK is a high-level programming framework that alleviates resource management and the deployment of complex applications on heterogeneous compute clusters. The PTK framework supports the development of task-based applications with Python and provides annotations that enable programmers to influence resource management, and specify performance requirements directly at the programming language level. It also facilitates the integration of these annotations into existing codebases. Behind the scenes, PTK relies on the widely-used Kubernetes container orchestration framework to deploy Python applications on clusters, either on-premises or in the cloud. The use of Kubernetes is transparent to the user, and all artifacts (e.g., manifest files) required for Kubernetes are automatically generated by PTK.

PTK facilitates the preparation of various deployment configurations on either single or multiple nodes, whether in cloud-based or on-premises infrastructures, while controlling the use of specific hardware for specific program tasks. The PTK framework provides users with high-level means required for improving the portability, performance, and scalability of compute- and data-intensive applications.

This paper provides the following contributions: (1) we propose the PTK framework, which automates the deployment of Python programs on on-premises or cloud infrastructures using the Kubernetes container orchestration framework; (2) we propose a set of annotations for Python to enable programmers to control how program tasks are mapped to containers and Kubernetes Pods, allowing users to quickly experiment with different deployment configurations; (3) we propose another set of Python extensions for specifying resource constraints and performance requirements, e.g., the number of CPUs or GPUs to be used for a certain programming task, or the maximum runtime of a task; (4) we develop a prototype implementation of the PTK framework and perform initial experiments on single cluster nodes with a real-world ML application to demonstrate some of its capabilities.

The remainder of this paper is structured as follows. Section II discusses related work. Section III introduces the PTK programming framework from an end-user's perspective, focusing on the programming support and Python annotations. Section IV outlines the implementation of the PTK framework and describes the generation of Kubernetes artifacts and the deployment of applications. Section V reports on initial experiments with an ML application for leaf classification. Finally, Section VI concludes the paper and provides an outlook on future work.

II. Related Work

Challenges in efficiently utilizing resources, achieving scalability, and reducing the costs of compute- and data-intensive applications have led to the development of numerous tools and frameworks. One of them is Kubernetes, an open-source, widely used orchestrator that facilitates the deployment, maintenance, scaling, and portability of applications for various cluster environments, e.g., cloud-based and on-premises [6]–[8]. PTK uses Kubernetes' smallest deployable unit — a Pod. Pods allow for the use of different physical nodes and serve as the encapsulating entity for a collection of containers, which are packages with everything needed to run the code [9]–[11].

Using Kubernetes requires a lot of low-level infrastructure-as-code (IaC) work, including defining storage solutions, creating images, managing resources, and configuring pod and container communication. This low-level code is automatically generated by PTK, which is one of its main benefits.

There are other tools based on Kubernetes, e.g., Kubeflow, which is an open-source project that aims to simplify the deployment of ML workflows on Kubernetes [12], [13]. It is based on Python, like the PTK framework, and allows the deployment of ML pipelines to the cloud. With a broader spectrum of tools, it is a comprehensive solution for generating ML models.

Martín et al. (2022) [14] proposed Kafka-ML, which allows the deployment of ML models represented as workflows using a combination of event-driven architecture and containerization, closely related to the PTK framework. However, we aim to work with cluster solutions using Python, whereas the authors focus on a single-node solution with a user interface representation for deployment.

An article presented by Balla et al. (2020) [15] deepened our understanding of the processes involved in scaling containerized applications using Kubernetes. In their paper, the authors introduced an adaptive technique and an application that enables the system to automatically scale resources both horizontally and vertically. Similarly, Shan et al. (2023) [16] introduced an adaptive resource allocation scheme for containerized workflows aimed at improving resource utilization.

Another example tool is Faasm [17], a serverless platform for running high-performance and portable serverless applications, allowing functions to share regions of memory with low-latency concurrent access to data. This platform can be deployed on Kubernetes, using its orchestration capabilities for operating serverless functions.

III. PTK Programming Framework

In this section, we provide an overview of the PTK framework, outline the concept of tasks and the proposed task-based programming style, and introduce different Python annotations for influencing the deployment configuration, including resource and performance requirements and constraints.

A. Overview

Figure 1 provides a high-level overview of the PTK framework. PTK takes as input a Python application with annotations and generates all the artifacts required for deploying the application on an existing Kubernetes cluster. The generated artifacts include manifest files with various Kubernetes objects, as well as Docker container images for the independent execution environment of the tasks within the program. The Kubernetes objects supported by PTK include Deployments (the main block for deploying any application to the cluster) and Volumes/Persistent Volume Claims (PVC) (storage abstractions for retaining data).

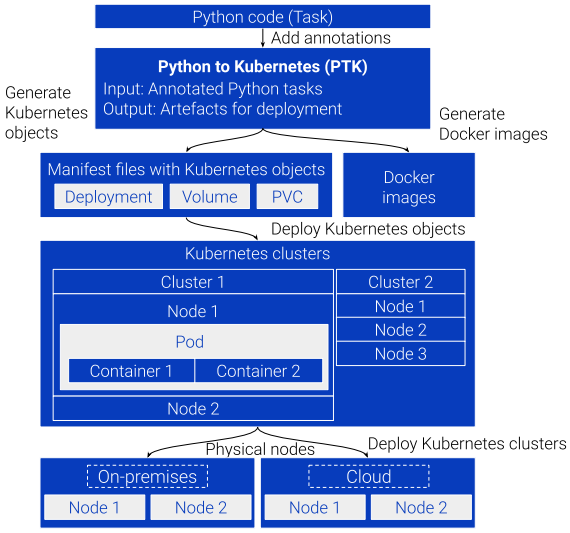

Fig. 1. Overview of the Deployment Schema with the PTK Framework

A major aspect of PTK is its explicit support for the configuration of the generated Kubernetes deployment, allowing the programmer to influence how the tasks of a program should be packed in containers and how these containers should be organized into Kubernetes Pods.

By leveraging PTK as an abstraction layer over containerization mechanisms, we can deploy applications to either cloud environments, such as a single cluster on Google Cloud Platform (GCP), or on-premises. PTK has the capability to work with Pods and the containers within them, generating finely-tuned Kubernetes manifest files tailored to these environments.

B. Task-based Programming

The PTK framework requires applications to be written using a task-based programming approach [18]–[21]. The user has to designate the most important functions in their Python code as tasks using PTK annotations. Internally, task-based applications are represented as a directed acyclic graph (DAG), where nodes represent tasks and edges usually represent data dependencies between tasks, i.e., one task generates data that is consumed by another task [22]. Such representations are particularly well-suited for applications whose tasks can be organized as a pipeline [23].

For a task, different implementation variants may be provided by the programmer (or taken from libraries) to efficiently utilize different target architectures. For example, a task may have a multi-threaded implementation variant that can make use of multi-core CPUs or an implementation variant written in CUDA for targeting GPUs. While such implementation variants are managed by the PTK framework, the user can influence which variant should be used on a specific target architecture by means of annotations. By designating information about resource requirements directly in Python code, users can manage configurations for each implementation variant. This information is then converted by PTK to prepare tasks for deployment.

C. Leaf Classification ML Pipeline

To demonstrate the capabilities of the PTK framework, we selected a Python-based visual categorization pipeline designed to generate ML models for identifying foliar disease categories in tomato leaves from images [24]. We transformed this application into a task-based DAG comprising five nodes: Download, Prepare, Preprocess, Train, and Evaluate. Each node corresponds to a PTK task (see Fig. 2).

The dataset comprises 10 classes of images, with 9 classes representing specific types of diseases and the 10th class indicating a healthy state. The following code excerpt shows the corresponding Python code of the application:

```
def download_data():
    ...
    return download_results
...
def preprocess_data(prepare_results):
    ...
    return preprocess_results
```

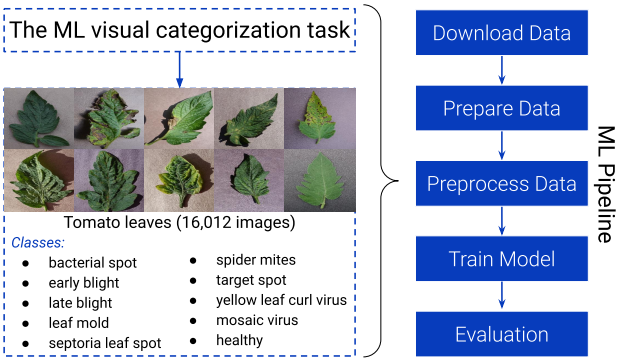

Fig. 2. Visual Categorization ML Pipeline

```
def train_data(preprocess_results):
    ...
    return train_results

def eval_data(train_results):
    ...
    return eval_results

def main():
    download_r = download_data()
    prepare_r = prepare_data(download_r)
    ...
```

We observe distinct Python functions or tasks, where each task returns data as a result, which eventually will be used as arguments for the next task. For example, `preprocess_data` returns `preprocess_results`, which contains data for generating the ML model in `train_data`. The ML pipeline contains both compute-intensive parts, such as `train_data`, which requires a large amount of computational resources for generating ML models, and data-intensive parts, such as `prepare_data`. The data transmitted between different blocks also varies in terms of size and content. For instance, the dataset transmitted between `prepare_data` and `train_data` for training is bigger than the dataset transmitted between `train_data` and `eval_data`, which contains the created ML model and the small dataset for evaluating the model.

D. Deployment Configuration Annotations

Using PTK annotations, users can influence the deployment configuration of an application, i.e., how the tasks of an application should be organized into containers and Pods. The following annotations are provided:

1) Task Annotation: The task annotation `@task(name='task1')` is used to designate that a Python function will become a PTK task managed by the PTK framework. By default, each PTK task is put into a separate container and deployed in a separate Pod. Note, if a Python function has the pod and/or container annotation (see below), it will become a PTK task by default.

2) Container Annotation: The container annotation `@container(name='cont1')` is used to designate that a task will be put into a container named 'cont1'. The container

annotation can be used to place multiple functions into the same container.

3) Pod Annotation: The pod annotation `@pod(name='pod1')` is used to designate that a PTK task will be put into a pod named 'pod1'. By adding `@pod` with the same name to different functions, multiple functions will be grouped into the same Pod, with each function running in a distinct container.

This approach can reduce resource and network overhead, simplify deployment, and facilitate horizontal scaling since the entire Pod, including all its containers, can be replicated across nodes. If only `@container` is specified and no `@pod` is provided, each function will be placed in its own Pod by default, with each Pod containing only one container.

Pods versus Containers: Containers are executable packages that include all required libraries, environments, and settings to run the code. They are isolated from each other within one operating system (OS). Kubernetes Pods, on the other hand, are a higher level of abstraction over containers, representing an additional virtualization level. They are the smallest independently deployable units, allowing the use of networking for communication across different nodes of a cluster. Deploying an application with multiple Pods allows the use of different cluster nodes and thus opens opportunities for horizontal scaling. In contrast, containers are restricted to within a single pod, serving as the encapsulating entity for a collection of tasks. Additionally, each Pod can have its own separate storage system within Kubernetes. Containers within a Pod share the storage defined for that Pod [10], [11].

For Kubernetes clusters, there are several approaches for configuring deployments, which are covered by PTK:

1) Pod with Many Containers (Each Container with One Task): Sharing the same OS by deploying tasks in different containers inside one Pod offers the opportunity for high-speed data transmission. Implementing two tasks as containers inside one Pod offers a more lightweight solution than implementing tasks as separate Pods, because the deployment has only one additional level of abstraction instead of two. This approach still allows vertical scaling for each container separately; however, it presents challenges for horizontal scaling, as Kubernetes replicates the Pod with all containers inside it.

2) Pod with One Container and One Task: Pursuing a high level of horizontal scalability, where each task can be scaled separately, users need to implement each task in a separate container and place these containers in separate Pods. In Kubernetes, each Pod, along with a task, can be replicated across different cluster machines, performing simultaneously.

3) Pod with One Container and Many Tasks: Alternatively, users can place all tasks inside one container to achieve better performance, representing the application as a monolith. However, this approach does not allow for the ability to scale individual tasks of a pipeline horizontally by using multiple nodes, or vertically by allocating more resources.

PTK eases the process of creating different deployment configurations of applications through annotations.

E. Data Management and Task Communication

Tasks in the considered applications communicate with each other by transmitting data. Upon completing one task, the results are retrieved by the subsequent task for further processing. With PTK, users do not need to change functions to implement data transmission mechanisms between them. By implementing annotations, it is possible to use mechanisms already created by PTK, which transform the input and output of each function internally to perform data transmission between tasks after deployment. PTK supports the following options for communication:

1) Volumes/PersistentVolumeClaim (PVC): This approach generates storage for communication between `@pod`, `@container`, or `@task`. PTK supports the following mechanisms: (1) Volumes, where each volume is tied to a specific Pod, and its lifespan equals the Pod's lifespan. When a Pod is terminated, its volume is also deleted; (2) PVCs, which are not tied to specific Pods and have independent lifecycles, allowing multiple Pods to share the same storage.

```
from ptk import task

@task(name='download',
      output={'type':'PVC','size'=20Gb'})
def download_data():
    ...
    return download_result

@task(name='prepare',input={'download'})
def prepare_data(download_result):
    ...
```

In this code, two tasks, `download_data` and `prepare_data`, are deployed in separate Pods. For data transmission between these tasks, we use a PVC with a size of 20GB, which is declared with `download_data`. `prepare_data` receives access to this PVC by adding the input for the task as the name of the `download_data` task ('download'), and adding arguments to the `prepare_data` function with the same name as the `return` value for `download_data`.

The PTK framework handles retrieving the data by connecting to the PVC, and waits until the data is fully uploaded to the location before proceeding to the next step. The `download` Pod is connected to a PVC, where the `download_data` task uploads all downloaded data to the PVC as a result. The `prepare_data` Pod waits for the completion of all data uploads from the `download_data` task.

2) Inter-process communication (IPC): IPC is used for communication between containers in one Pod.

```
@pod(name='main-dag',output={'type':'IPC'})
def download_data():
    ...
    return result

@pod(name='main-dag',input={'download_data'})
def prepare_data(download_result):
    ...
```

In this case, `download_data` and `prepare_data` are deployed in the same Pod `main-dag`, but in different containers with names matching the function names, which allows the use of IPC. Users do not need to change the code in this case, only the output type.

3) Direct function calls: This method is used for data transmission between functions inside one container, where one function is called by another function.

F. Resource Management

PTK allows users to specify resource requirements for tasks, containers, and Pods in the form of 'limits' and 'requests' for virtual CPUs (vCPU), GPUs, and RAM. Requests specify the minimum resources required for deployment, while limits contain the maximum allocatable resources. They are used for vertical scaling, allocating more or less resources for containers within one Pod. At the same time, Pods are used for horizontal scaling of applications. Kubernetes provides mechanisms for deploying several replicas of the same Pod to different nodes.

1) Task Annotation: As the PTK framework works with tasks organized into containers and Pods, it allows users to tune resource management and replication for each task:

```
@task(cpu_requests=6,cpu_limits=8,
memory_requests='8Gi',memory_limit='16Gi',
gpu_requests='1',gpu_limits='4',num_replicas=2)
prepare_data(...):
```

The `num_replicas` argument defines the number of replicas used for horizontal scaling across Kubernetes clusters. For example, `num_replicas` can be used for ensemble ML, when users can create multiple small models on different nodes using different subsets of the training data. As a result, it combines the predictions of multiple models to produce a single final prediction [25]. Each replica requires 6 vCPUs, 8 GiB of RAM, and 1 GPU, with the maximum resources that might be allocated being 8 vCPUs, 16 GiB of RAM, and 4 GPUs.

2) Container Annotation: This annotation allows to use only 'limits' and 'requests', i.e. `cpu_requests`, `cpu_limits`, `memory_requests`, `memory_limits`, `gpu_requests`, and `gpu_limits` arguments.

3) Pod Annotation: Whereas `@task` covers all opportunities for resource management, represented as a step above Kubernetes logic, `@pod` allows to use only `num_replicas`.

IV. Transformation to Kubernetes

In this chapter, we outline how the PTK framework automates the process of transforming Python programs with PTK annotations, so that they can be executed on a Kubernetes cluster. This process includes the generation of container images, Kubernetes objects, and manifest files, as well as scripts for deploying and running a PTK application.

A. Generation of Container Images

One of the required operations for deployment is the containerization of annotated tasks before their deployment. The PTK framework automatically containerizes tasks, generates images, and deploys them.

This operation occurs during the main process of generating Kubernetes manifest files, where the created image is added to the manifest. If we need to create an `image` of the task during the generation of Kubernetes manifest files, the PTK framework can generate a simple Dockerfile, add the path to the task, and name the image based on the concatenation of the file name and the task name. The framework then generates the Docker image and deploys it to Docker Hub [26]. However, this option is currently restricted to bare Python applications with default libraries and TensorFlow-based applications.

B. Generation of Containers and Pods

With the current version of PTK, we assume that the target cluster for deployment is already created. From the cluster, we need to obtain data on how many resources are available for allocation to the application, based on which we can add non-functional requirements for tasks [27], [28]. Additionally, we need to decide on the configurations of Pods and containers for applications. For example, should we use only one container and one Pod to host all tasks, or do we need to use a separate Pod and container for each task?

The PTK framework processes the source code annotations as follows:

(i) For each `@pod` annotation of a task, the framework either inserts a new Pod or updates an existing one. Moreover, PTK adds a declared `@container` for this task or automatically creates one.

(ii) When `@container` is created and used along with `@pod` for a function, PTK adds the container to the specified Pod and includes this Pod into the internal PTK structure. If no `@pod` annotation is provided, PTK automatically creates a new Pod for the `@container`.

(iii) The `@task` annotation behaves like a combination of both a `@pod` and a `@container`, automatically adding the corresponding Pod and container to the PTK internal structure.

These annotations highlight the simplicity of adding a few lines of code compared to manually crafting the structure of Kubernetes manifest files for deployment. The manifest files are automatically generated based on the provided data.

C. Data Transmission

PTK works with three types of communication: Volumes/PVC, IPC, and Direct function calls. Using our framework, users do not need to change the code of tasks or implement data transmission between them by themselves.

1) Volumes/PVC: According to the example in Section III-E, the producing data task contains an `output` argument. During image generation, this argument allows the framework to find and substitute the `return download_result` of the function with an internal method that preserves data into a PVC. For the generation of the manifest file, it creates a Volume, a PVC, and attaches this PVC to the Pod. The consuming data task contains the `input='download'` argument, which includes the name

of the producing task, as well as the `download_result` argument with the same name as the return value in the `download_data` function. This enables PTK to automatically get this data from the PVC.

2) IPC: This option follows the same logic. By adding `output` and `input` for two functions, PTK internally substitutes the return value of the producing functions and the arguments of the consuming function with IPC data communication.

3) Direct function calls: Users need to place the functions in one file and add the call of one function into the other. PTK generates the image and Kubernetes manifest by combining all these functions, treating them as one. For this purpose, it scans the task to find other function calls located in the same file.

D. Deployment and Execution of PTK Applications

After creating the Kubernetes cluster, implementing the desired Pod-container configuration using annotations, adding non-functional requirements, and resolving issues in the creation of container images, the PTK framework can generate Kubernetes manifest files.

PTK uses preprocessing mechanisms for the generation of Kubernetes manifest files. Users need to call the annotated task within their Python code, and the PTK framework will generate the manifest files before running the main Python code of the task. If you have multiple annotated tasks to be run in one session, the framework gathers and analyzes the data from all annotations during this preprocessing phase before running these tasks. Additionally, to prevent the local running of pipelines, PTK has an option to stop the execution of the main Python code after analyzing all annotations, and creating all Kubernetes manifest files before running the main code.

Internally, the framework transforms annotations into its own structure. The main PTK structure is represented as a list of Pods, which contains the main information from the `@pod` annotations, the file where the `@pod` is located, and the task where it is added. Each Pod in the list also has containers and volumes/IPC with their information, including the tasks for which they are declared. This information simplifies distinguishing between Pods when multiple Pods are created during the same session.

Each Pod is then transformed into a Kubernetes Deployment object, and all Pods are saved in their own files, with names based on the concatenation of the Pod name and the names of the tasks for which the Pod's containers are created. Once the manifest file exists, its regeneration is triggered.

The created manifest files are dependent on the target machine, considering available resources. Users need to ensure that the machine's resources are sufficient and meet or exceed the specifications in the non-functional requirements of their application. After creating the cluster and obtaining the Kubernetes manifest files, users can deploy these files to the cluster and run their pipeline on the target infrastructure.

V. Evaluation

For the initial experimental evaluation of the PTK framework, we decided to use the Python-based ML pipeline from Section III-C. All experiments were conducted on GCP using the us-west1-c zone. One of the main benefits of Kubernetes is the ability to use multiple nodes within a cluster. However, there are cases when the data size and calculation complexity of the pipeline allow it to be deployed on a single node, which increases the application's speed by eliminating the need to transmit data between nodes. Therefore, all our experiments were conducted using a single node.

For the first experiment, we used the PTK framework to generate Kubernetes manifest files and deploy three different Pod-container configurations of this application. Our goal was to determine the impact of each configuration on the execution time for generating the ML model and the associated costs. For the second experiment, we deployed the best configuration identified in the first experiment to different nodes, in order to show the influence of different machines on the time for generating the ML model and the cost.

A. ML Application and Different Pod-Container Configurations

One of the major challenges for the application was to determine a good configuration in terms of how tasks are organized into containers and Pods. Using `@task` annotations, we implemented Configuration 1 of Pod-container packaging, which we consider the default, i.e., each task goes into a separate container and a separate Pod. We chose to represent the pipeline as four Pods, each containing one container. PTK internally generated separate Kubernetes manifest files, one for each Pod. In turn, each container holds one task of the ML pipeline (see Fig. 3). We assume that the data has already been downloaded using the Download task.

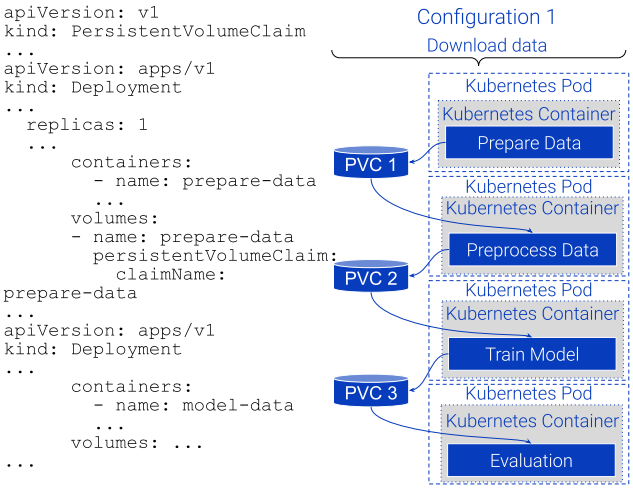

Fig. 3. Main Pod-container Configuration for the ML Pipeline

PTK creates images for each task. For communication between different steps, we use PVCs, where one step uploads the data to the PVC and the next step downloads this data for further calculations, as described in Section III-E.

In addition to Configuration 1, for the same pipeline, we developed two other configurations using `@pod` and

`@container` annotations (see Fig. 4). Configuration 2 uses one Pod with multiple containers, where each container contains one task. Configuration 3 uses one Pod with one container for all tasks. Communication between tasks is also accomplished with PVCs, as in Configuration 1.

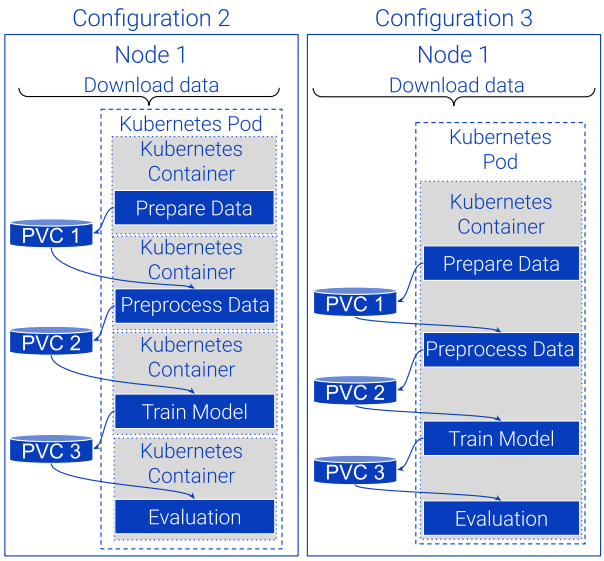

Fig. 4. Configurations 2 and 3 with Different Pod-Container Groupings for the ML Pipeline

For the experiments, we deployed these configurations on single-node clusters and compared the execution time and cost for each configuration (see Table I).

TABLE I
EXECUTION RESULTS OF DIFFERENT CONFIGURATIONS WITH EQUAL RESOURCE CHARACTERISTICS

Configurations	**#1**	**#2**	**#3**
execution_time, secs	70.432	68.734	64.234
execution_time, %	109.6	107	100
node_price_hr, $	0.047		
gpu_price_hr, $	0.355		
zone_price_hr, $	0.101		
total_cost, $	0.0099	0.0096	0.009

We used an n1-standard-1 node with 1 CPU and 3.75 GB of RAM. Additionally, we added an NVIDIA Tesla T4 GPU with 16 GB of memory. We defined resource requests using the PTK framework. For Configurations 1 and 2, each task required 0.25 vCPU and 1 GB of RAM, except for the Evaluation task, which required 0.75 GB of RAM, and the Train task, which used the GPU. For Configuration 3, we allocated all the resources of the node to one container. The corresponding cost for generating the ML model was calculated based on GCP pricing resources [29], [30].

Changing from Configuration 1 to Configuration 3 by decreasing the number of Pods and containers reduced both the execution time and cost by roughly 10%. This reduction is due to the fewer number of containers and Pods that Kubernetes needs to maintain, which allocates resources away from the execution of the pipeline itself (see Fig. 5).

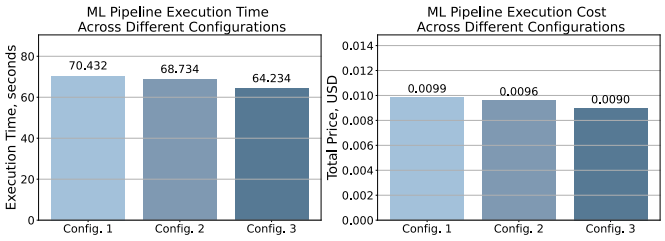

Fig. 5. ML Pipeline Execution Time and Cost Across Different Configurations

B. ML Applications and Different Resource Characteristics

Selecting Configuration 3 as the fastest and cheapest, we compared its deployment on three types of nodes, measuring the time required to create the ML model, and calculating the costs based on the resources used.

We performed three experiments, one for each node (see Table II). Scenario 1 reflects the results of Configuration 3 from Table I. For Scenario 2, we used the same node but without the GPU. For Scenario 3, we used an n1-standard-2 machine with 2 CPUs and 7.5 GB of RAM. We utilized all the resources of the nodes for each scenario.

TABLE II
EXECUTION RESULTS OF CONFIGURATION 3 WITH DIFFERENT RESOURCE CHARACTERISTICS

Scenarios	**#1**	**#2**	**#3**
node_type	n1-standard-1		n1-standard-2
execution_time, secs	64.234	1725.286	957.408
node_price_hr, $	0.047		0.095
gpu_price_hr, $	0.355	-	-
zone_price_hr, $	0.101		
total_cost, $	0.009	0.071	0.052

It is evident that the use of GPUs significantly improves performance. When the GPU was not used for the Train task, the time for generating the ML model increased by almost 27 times, from 64 to 1,725 seconds (see Fig. 6). Additionally, using two CPUs in an n1-standard-2 node decreased the time by 1.8 times, as the Train task could utilize both CPUs to speed up the creation of the ML model.

The cost is correlated with the execution time; however, for such experiments, the overall cost increase was less than the increase in execution time because Scenarios 2 and 3 used less expensive nodes. As you can observe, deploying the application using only the CPU in Scenario 2 is almost 8 times more expensive than using the GPU in Scenario 1, even though the price of the node in Scenario 1 is 3.38 times higher than in Scenario 2. The same situation is observed between Scenario 2 and Scenario 3, where the price of the node in Scenario 3 per hour is 1.32 times higher, but the total execution cost is 1.37 times lower.

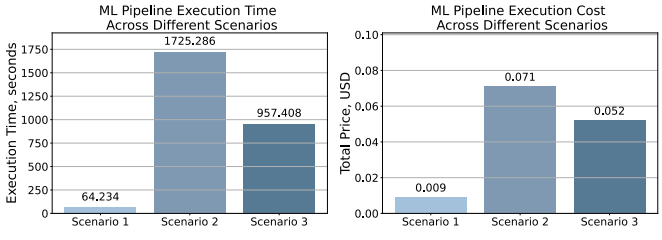

Fig. 6. ML Pipeline Execution Time and Cost Across Different Scenarios

VI. Conclusion

In this article, we proposed PTK, a high-level Python-based programming and resource management framework. It automates the deployment of applications on both on-premises and cloud infrastructures using Kubernetes. We proposed annotations for Python, allowing users to specify how the main tasks of an application should be configured into containers and Pods. In addition, PTK offers annotations for specifying resource requirements. We developed a prototype of PTK and demonstrated the framework's utility through initial experiments, involving the implementation and deployment of a real-world ML application with various resource configurations.

For future steps, the development of the PTK framework will include work on the automatic provisioning of Kubernetes clusters based on the user-specified resource requirements. Moreover, we plan to introduce annotations for enabling users to specify performance requirements and expectations for tasks, e.g., limits for the execution time or the price, and to use this information in order to select appropriate resources. Future work will also focus on more extensive evaluations, including multi-node deployments, to further validate and refine the framework.

References

[1] P. Beckman, J. Dongarra, N. Ferrier, G. Fox, T. Moore, D. Reed, and M. Beck, "Harnessing the computing continuum for programming our world," Fog Computing: Theory and Practice, ch.7, Wiley, 2020.

[2] R. Miceli, G. Civario, A. Sikora, E. Cesar, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin, "AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications," Proceedings of the 11th International Workshop on the State-of-the-Art in Scientific and Parallel Computing (PARA 2012). Helsinki, Finland, June 2012.

[3] M. Kleppmann, Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems, "O'Reilly Media, Inc.", 2017.

[4] V. Amaral, B. Norberto, M. Goulao, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, H. Karatza, C. Kessler, P. Kilpatrick, H. Martiniano, I. Mavridis, S. Pllana, A. Respicio, J. Simaon, L. Veiga, and A. Visa, "Programming Languages for Data-Intensive HPC Applications: a Systematic Mapping Study," Parallel Computing, Volume 91, March 2020, Elsevier.

[5] S. Benkner, A. Arbona, G. Berti, A. Chiarini, R. Dunlop, G. Engelbrecht, A. F. Frangi, C. M. Friedrich, S. Hanser, P. Hasselmeyer, R. D. Hose, J. Iavindrasana, M. Koehler, L. Lo Iacono, G. Lonsdale, R. Meyer, B. Moore, H. Rajasekaran, P. E. Summers, A. Wöhrer, and S. Wood, "@neurIST - Infrastructure for Advanced Disease Management through Integration of Heterogeneous Data, Computing, and Complex Processing Services," IEEE Transactions on Information Technology in Biomedicine; 14(6):1365-77, 2010.

[6] V. Medel, R. Tolosana-Calasanz, J.Á. Bañares, U. Arronategui, and O.F. Rana, "Characterising resource management performance in Kubernetes," Computers & Electrical Engineering, vol.68, pp.286-297, 2018.

[7] J. Rosso, R. Lander, A. Brand, and J. Harris, Production Kubernetes, "O'Reilly Media, Inc.", 2021.

[8] B. Burns, J. Beda, K. Hightower, and L. Evenson, Kubernetes: up and running, "O'Reilly Media, Inc.", 2022.

[9] C. Anderson, "Docker [software engineering]," IEEE Software, vol.32, no.3, pp.102-c3, 2015.

[10] K.T. Seo, H.S. Hwang, I.Y. Moon, O.Y. Kwon, and B.J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," Advanced Science and Technology Letters, vol.66, pp.105-111, 2014.

[11] J. Dobies, and J. Wood, Kubernetes operators: Automating the container orchestration platform, "O'Reilly Media, Inc.", 2020.

[12] E. Bisong, Building machine learning and deep learning models on Google cloud platform, Berkeley, CA: Apress, 2019.

[13] C. Xu, G. Lv, J. Du, L. Chen, Y. Huang, and W. Zhou, "Kubeflow-based automatic data processing service for data center of state grid scenario," 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), pp. 924-930. IEEE, 2021.

[14] C. Martín, P. Langendoerfer, P.S. Zarrin, M. Díaz, and B. Rubio, "Kafka-ML: Connecting the data stream with ML/AI frameworks," Future Generation Computer Systems, vol.126, pp.15-33, 2022.

[15] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, pp.1-5. IEEE, 2020.

[16] 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, n.3, pp.723–743, 2023.

[17] "Faasm," GitHub, accessed May 20, 2024, https://github.com/faasm/faasm

[18] M. Niu, B. Cheng, Y. Feng, and J. Chen, "Gmta: A geoaware multi-agent task allocation approach for scientific workflows in container-based cloud," IEEE Transactions on Network and Service Management, vol.17, no.3, pp.1568-1581, 2020.

[19] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," IEEE Transactions on Parallel and Distributed Systems, pp.1-14, 2017.

[20] S. Benkner, S. Pllana, J. L. Traeff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems," IEEE Micro, vol. 31, no. 5, pp. 28-41, Sep./Oct. 2011, doi:10.1109/MM.2011.67.

[21] O. Aumage, P. Carpenter, and S. Benkner, "Task-based performance portability in HPC," accessed May 21, 2024, https://www.etp4hpc.eu/news/273-task-based-performance-portability-in-hpc.html

[22] R.A. Sahner, and K.S. Trivedi, "Performance and reliability analysis using directed acyclic graphs," IEEE Transactions on Software Engineering, vol.SE-13, n.10, pp.1105–1114, 1987.

[23] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault, "High-Level Support for Pipeline Parallelism on Many-Core Architectures," Proc. European Conference on Parallel Computing, Euro-Par 2012, Rhodos, Greece, Aug. 27-31, 2012, LNCS 7484, pp. 614-625, Springer Verlag.

[24] H. Hapke, and C. Nelson, Building machine learning pipelines, "O'Reilly Media, Inc.", 2020.

[25] T.G. Dietterich, "The handbook of brain theory and neural networks," MIT Press: Cambridge, MA, 2002.

[26] "Docker Hub," accessed June 4, 2024, https://hub.docker.com/.

[27] L. Chung, and J.C.S. do Prado Leite, "On non-functional requirements in software engineering," Conceptual modeling: Foundations and applications: Essays in honor of John Mylopoulos, vol.5600, 2009.

[28] M. Glinz, "On non-functional requirements," 15th IEEE international requirements engineering conference (RE 2007), pp.21–26. IEEE, 2007.

[29] "VM instance pricing," Google Cloud, accessed May 20, 2024, https://cloud.google.com/compute/vm-instance-pricing.

[30] "Google Cloud's pricing calculator," Google Cloud, accessed May 20, 2024, https://cloud.google.com/products/calculator?hl=en.