# RAPTURE: a Remotely Accessible Platform of Testbeds for UAS Research and Education

Justin S. Lee*, Nicholas Palmer†, Junfei Xie‡, Yan Wan§, Kejie Lu¶, Shengli Fu‖

**The development of unmanned aerial systems (UASs), a.k.a., drones, is a highly cross-disciplinary subject, demanding diverse expertise to build comprehensive systems from scratch. The introduction of open and standardized hardware testbeds has the potential to accelerate UAS development and mitigate the barrier to entry in this rapidly growing field. In this paper, we introduce an open platform based on Kubernetes that offers users remote access to a shared network of hardware testbeds consisting of UASs, ground rovers and compute devices. Three use cases are presented to showcase the key features of the proposed platform and its potential to facilitate research and education in UAS, robotics and edge computing.**

## I. Introduction

In recent years, we have witnessed rapid progress and increasing interest in unmanned aerial system (UAS) development within both industry and academia. UASs have been successfully applied to wildfire management [1], package delivery [2], crop monitoring [3], search and rescue [4], and several other niche uses. Recent advances in automation, computer vision, and artificial intelligence offer computationally complex solutions to solve problems in these applications.

Within this increasingly advanced computing context of UAS applications, much of the emerging research is focused on software solutions [5, 6] rather than enhancing UAS hardware design. For example, a user developing aerial photogrammetry may only need a generic drone to get their system airborne along with a generic set of sensors and computing resources to support their application. Many past studies have predominantly relied on simulations to evaluate these software-oriented solutions–which can be largely attributed to the difficulties and expenses associated with building a UAS hardware testbed from scratch as well as the relatively recent FAA regulations on conducting flight tests. Nonetheless, hardware testing is crucial for thorough system validation. This transition from simulated testing to hardware testing remains a major hurdle in UAS development.

We imagine a possible solution to ease this crucial transition to hardware testing could be the availability of a public and remotely accessible multi-UAS hardware testbed, allowing users to test their software solutions on actual hardware while avoiding the difficulties traditionally associated with such an endeavor. Yet, there has been limited work in this direction. One related initiative can be found in the FIT IoT-Lab [7], which is an open hardware platform geared toward research in wireless communications technologies. The platform hosts thousands of nodes in various hardware configurations that users can execute experiments on. However, the FIT IoT-Lab concentrates on RF validation through embedded systems rather than a more robotics focused UAS platform as we propose. In previous studies [8–12], our lab developed a flexible and extensible networked airborne computing testbed that consists of multiple UAS with full functionality. In this paper, we aim to enable remote access to our testbed, allowing users around the world to test their UAS software solutions on realistic hardware testbeds. This will greatly lower the barrier to entry for participating in UAS development, by eliminating the necessity for developers to invest in or construct their own testbeds.

To facilitate users to remotely conduct experiments on our UAS testbed, containerization is a promising technique particularly considering the success of the state-of-the-art container orchestration software Kubernetes [13] in the cloud computing industry. Containerization in the cloud computing infrastructure has become an integral factor in the trend toward portable micro-service oriented computing and is beginning to be adopted in more edge-centric areas like UAS both in industry and among the research community. For example, the authors in [14] developed "tactical cloudlets", which are mobile cloud computing resources situated near the edge to support computation offloading and data staging from handheld edge devices. They demonstrated improvements in overall performance, energy efficiency,

*Undergraduate Student, Department of Electrical and Computer Engineering, San Diego State University, AIAA Student Member

†Undergraduate Student, Department of Electrical and Computer Engineering, San Diego State University, AIAA Student Member

‡Associate Professor, Department of Electrical and Computer Engineering, San Diego State University, AIAA Senior Member, Corresponding Author, email: jxie4@sdsu.edu

§Distinguished Professor, Department of Electrical Engineering, The University of Texas at Arlington, AIAA Associate Fellow

¶Professor, Department of Electrical and Computer Engineering, University of Puerto Rico at Mayagüez

‖Professor and Chair, Department of Electrical Engineering, University of North Texas

and availability through the adaptation of cloud computing techniques to enhance edge networks. More recently in [15], Red Hat describes their collaboration with Lockheed Martin utilizing the MicroShift Kubernetes distribution to handle AI computer vision workloads on-board the small Stalker military UAS platform. These examples exemplify the benefits of decentralizing computing by pushing it further to the edge, where data requiring processing and decision-making originates, while still maintaining a contiguous connected network through the adoption of cloud computing principles.

Recognizing the capability of containerization and its management through Kubernetes to facilitate robust concurrent execution of UAS software applications, this paper introduces RAPTURE, a Remotely Accessible Platform of Testbeds for UAS Research and Education. RAPTURE incorporates Kubernetes to deploy application containers on our hardware testbeds. Through a user-friendly web portal, RAPTURE allows researchers and educational users around the world to upload their software solutions, conduct remote testing on our hardware testbeds, and seamlessly gather experiment results. It is designed to support several key features:

- **Abstraction:** Containerization and container orchestration allows arbitrary environments to run on the cluster, supporting projects built for nearly all platforms and software stacks. Currently, Python is the supported language of choice.
- **Unified Results & Feedback:** Before experiments are run, users can specify what results are reported to the web portal.
- **Several Hardware Targets:** The platform supports various hardware targets, including pure compute devices, ground rovers, and UASs, supporting research in not only UAS, but also edge computing, robotics, and more.

In the rest of this paper, we first present the system design for RAPTURE and then the user experience for uploading experiments in Sections II and II.D. Next, we explore three use cases to demonstrate the utility of our platform in Section III. In Section IV, we investigate the performance overhead introduced by our platform. Lastly, we conclude by summarizing our research and suggesting several avenues for future work in Section V.

## II. System Design of RAPTURE

Creating a platform capable of running arbitrary experiments on several different hardware platforms requires a specialized architecture that utilizes containerization and abstraction. We begin this section with an overview of the proposed RAPTURE system design. We then discuss how our platform adapts Kubernetes from a cloud-computing use case to one that allows RAPTURE to execute user-uploaded experiments. We then briefly describe the selection criteria for our chosen Kubernetes distribution. Finally, we explain how users can interact with our platform to upload experiments and gather results.

### A. Overview

Fig. 1 depicts the architecture of RAPTURE *. The first layer is the RAPTURE web portal, which serves as the interface for remote users to interact with our platform. Through the web portal, users can download results and upload their experiments (a set of Python applications) to be executed on our hardware testbeds. The next layer is the Kubernetes cluster, which manages deploying applications to the network's *nodes*, including compute devices, ground rovers, and UASs. We support these three hardware targets, and each are interoperable together over the network by users. Overall, the system can be categorized into three core components: the web portal, the Kubernetes backend, and the SMILE module. Each of these components is described in more detail in the following subsections.

#### *1. Web Portal*

The web portal provides control surfaces for users to submit experiments, download results, and in general operate on our platform remotely, where an experiment refers to a deployment of one or multiple applications. Users initially will create an account, and once approved, they can create experiments. The user may define this experiment to include any number of applications, executed on various configurations of node types and counts. For example, a user may want to run two applications on a single drone node and another application on a compute node. If an experiment is to be run on only compute nodes, the experiment can be processed and deployed directly to the platform without humans in the loop. This policy ensures experiments without hardware requirements are run promptly and no resources sit idle. However, for experiments that run on UASs or ground rovers, necessary human interventions require that they be executed only after approval from an on-site administrator who prepares the hardware for the experiment. We envision

*The code can be found at https://github.com/justinscottlee/smile-rapture

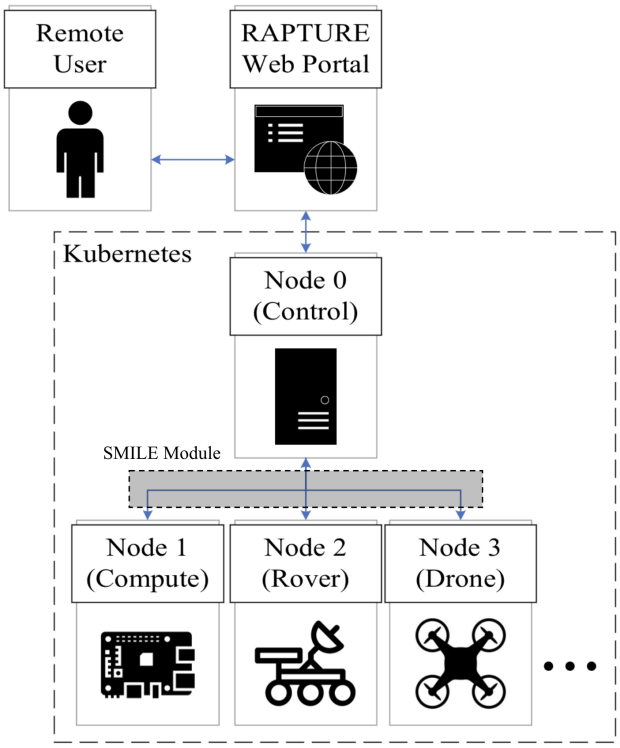


**Fig. 1 The architecture of RAPTURE.**

that such experiments will be queued and executed when a supervisor is available to monitor the runs.

Our platform currently supports applications written in Python. Expanding support to other languages is an intriguing avenue for future development, potentially enabling more performant applications. Even so, we have found that Python is a good option for its rich libraries in UAS and robotics research such as OpenCV for computer vision, PyTorch for machine learning, SciPy for computing, or ROS [16] for robot applications. We have minimized the development overhead for users to create Python applications that work on our platform. The following needs to be provided for each application: (1) the source files; (2) a requirements.txt file listing all Python module dependencies; (3) network ports the application listens for; and (4) specifications for which node and node type it should be allocated to within the hardware network.

*2. Kubernetes*

Kubernetes is a critical component of our platform, providing a virtualized layer for all user applications. In our implementation, a Kubernetes cluster consists of a set of real-world nodes, each capable of hosting containers. These containers are deployed to nodes within the cluster without needing awareness of the underlying configuration. Kubernetes provides abstractions for hardware and network communication through a virtual network. Our Kubernetes cluster consists of two node types: agents and a server node. The agent nodes, which include compute nodes, ground rovers, and drones, are where user applications are executed. The server node, or control node, manages the agent nodes. The web portal translates the user-provided application and node allocation into a deployment that the Kubernetes control node can distribute across the cluster. Since Kubernetes cannot run Python applications directly, the user applications must first be compiled into container images. Given the source files and requirements.txt provided by the user, the web portal can automatically generate the corresponding container images.

*3. SMILE Module*

The last core component of the platform is the SMILE module, which bridges the gap between the web portal and the agent nodes. It is a lightweight Python module that users can include in experiment code to send results and data back to the web portal. Result reporting is crucial for providing useful feedback in the form of text or arbitrary data.

The SMILE module provides logging functions that send data from the user code running in a container to the platform via the server node that runs with every experiment. The server node binds a ZeroMQ (REP) Reply pattern socket over TCP and awaits incoming data. When data is received and validated based on its type, the node performs a POST request to the web portal with the data, where it is processed and displayed to the user as the experiment runs.

### B. Adoption of Kubernetes-based System Hierarchy

Kubernetes is the most integral component in our management of applications on the hardware testbeds. We utilize three Kubernetes document types: Jobs, Services, and Namespaces. Each experiment is assigned a Namespace, a Kubernetes object that can contain other objects—in this case, Jobs and Services. Jobs hold the user's application in the form of a container. The containers created by these Jobs are fully sandboxed within their virtual environment, and need a Service object to communicate with each other. The Service exposes certain network ports for the container to be accessed by external applications. By using these three Kubernetes objects, we achieve the necessary functionality to run user experiments under Kubernetes. This hierarchy is depicted in Fig. 2.

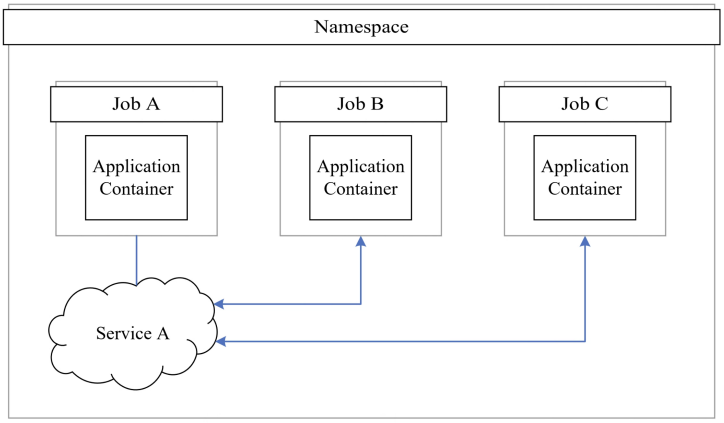


**Fig. 2 Hierarchy between Deployment, Pod, and Container within Kubernetes.**

This combination of Namespace, Jobs, and Services enables the execution of user applications. However, there still needs to be a way to interface these applications with the RAPTURE web portal so that users can interact with their experiments and retrieve results. We achieve this by instantiating a helper application alongside each user's experiment. User applications can then include the SMILE module described in Sec. II.A.3 with their applications to easily communicate with this helper application, which then forwards the data back to the web portal for the users to view.

### C. Selection of Kubernetes Distribution

There are many different Kubernetes distributions available, so making an informed selection of a suitable one for our application is vital. Numerous Kubernetes distributions are in production, each tailored to specific use cases. In this study, we evaluated four widely supported distributions: K8s, MicroK8s, MicroShift, and K3s.

K8s, or Kubernetes, is the standard upstream distribution of Kubernetes. It is commonly used in industry for cloud computing and is arguably the most powerful of these offerings. However, given that standard K8s isn't targeted at resource-constrained environments like UAS, we would prefer a more lightweight alternative. The other three distributions—Microk8s, MicroShift, and K3s—are all purpose-built to be lightweight and are targeted toward the Internet of Things (IoT) and edge devices.

MicroK8s is backed by Canonical [17], the company behind Ubuntu, and is available as a snap package for Linux environments. A double-edged feature of this distribution is its focus on modularity: features like name resolution within the cluster or a dashboard are disabled by default in an effort to make the distribution as minimal as possible. However, we find this modularity doesn't substantially improve resource usage compared to the other options that include these functionalities by default.

MicroShift is backed by Red Hat [15] and is targeted toward enterprise applications with edge devices. MicroShift

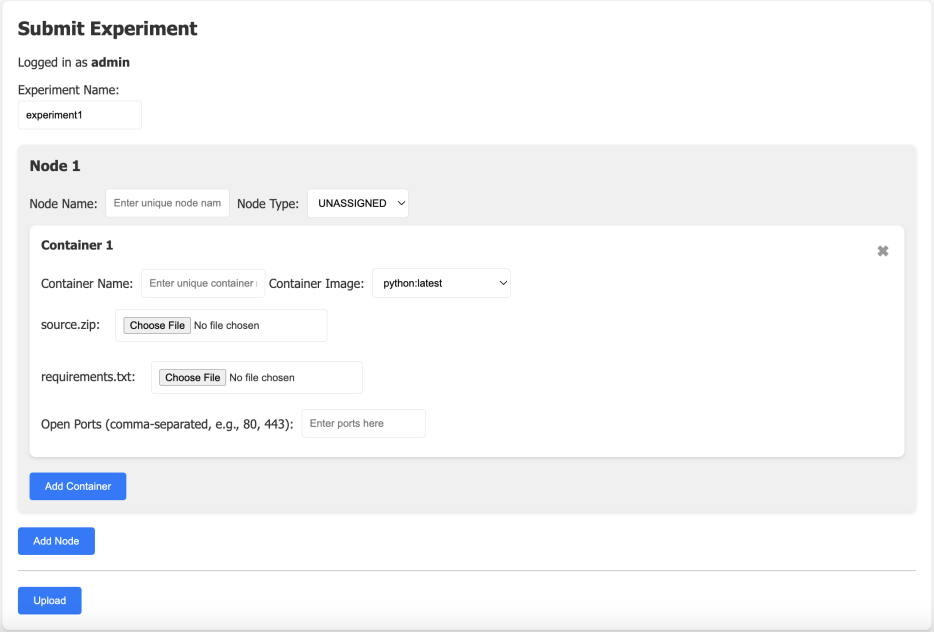


**Fig. 3 Screenshot of user interface used to upload experiments to the RAPTURE portal.**

is relatively new, so it is less mature than the other options and has limited support for different CPU architectures. However, it has seen adoption in industry for UAS applications; particularly in the collaboration between Lockheed Martin and Red Hat for the Stalker UAS platform.

K3s [18] is open-source and is also the most popular among the lightweight alternatives we have considered. Some of the key advantages of it are that it allows for single-command installation and cluster initialization, is packaged into a single binary, and also uses a more lightweight database than the other distributions.

After evaluating each of these Kubernetes distributions, we settled on K3s. The most valuable benefit of K3s is its ease of use, with installation and cluster initialization possible with a single command on a fresh device. This ease of use can aid the future growth of the platform when more UASs are added to the network, or if others want to recreate similar systems. Additionally, K3s's use of a more lightweight database has been found to result in nearly an order of magnitude lower disk utilization when compared to K8s and MicroK8s [19].

### D. User Experience

The development of the web portal for user interaction with our platform is a critical aspect that must be well-designed to ensure both usability and functionality. The user flow should be intuitive, allowing users to configure multiple containers and utilize various types of nodes simultaneously, while retaining all necessary functionalities. In this subsection, we present our user interface, which offers an accessible portal that meets these requirements and allows users to access results through the same interface.

Figure 3 illustrates the web interface that facilitates users in uploading their experiments to the network. This interface allows users to specify the number of nodes required for their experiment, including compute devices, ground rovers, and UASs, and to allocate specific applications to designated nodes. Upon uploading, the experiment is sent to the backend and queued for execution by the platform. The interface provides several customization options, such as unique naming for containers and nodes, enabling users to reference specific nodes from the SMILE module when reporting results or controlling hardware. Users can also choose container-based images from a predefined list, allowing them to run different versions of Python or other arbitrary images. By specifying open ports, users can enable communication between containers, facilitating inter-application traffic via a service. Additionally, specifying the node type allows users to select from various options, including pure compute nodes, rover nodes, or drone nodes.

Once an experiment is initiated—either by the user or a supervisor in the case of rover or drone runs—users need a method to visualize or transfer the results for analysis. Figure 4 depicts the experiment view, showcasing the results generated from the experiment. The platform provides a method for sending arbitrary results back to the portal when

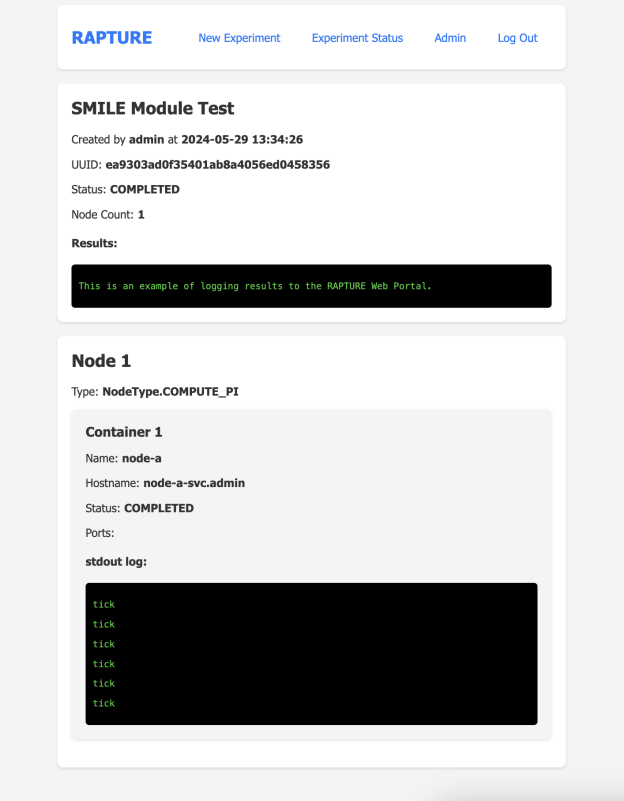


**Fig. 4 Screenshot of user interface displaying container status and results from the experiment run.**

users write platform code. Each container's standard output (STDOUT) is also captured and displayed to the user, serving as an additional data source for experiment analysis. This approach ensures that users can effectively monitor and analyze the outcomes of their experiments.

## III. Use Cases

To demonstrate the usage and value of our platform, we present three use cases that highlight its various capabilities [†]. The first use case showcases a multi-node collaborative computing challenge focused on computer vision. The second use case features the rover node and its deployment process, while the third illustrates a multi-node collaborative drone challenge.

### A. Case One: Satellite Image Demosaicing

Satellite image demosaicing is an important task requiring massively distributed systems because of the high resolution and volume of data collected. We develop a simple proof-of-concept OpenCV script that applies the divide and conquer paradigm to the problem and functions on our platform. Given an arbitrary number of images, we distribute them across multiple client nodes until an image pair is successfully stitched. The resulting image is then sent back to the server node to complete the stitching process.

Algorithms 1 and 2 describe the high level structure of the algorithm. The implementation uses OpenCV ORB feature matching and keypoint-descriptor matching to compute a homography transform for image pairs. It includes a server function, a client function, and a demosaic utility module. This modularity allows one container to run the server code, while an arbitrary number of clients run the client code, waiting for the server to distribute images across containers. Clients can run on the same node, or different nodes for different applications.

For demonstration purposes, we distribute 41 satellite images (see Figure 5) to five client containers running on a single node (Dell Precision 3630 with an Intel i7-8700K processor and 16 GB of RAM) for stitching. To control the

[†]Code is available on GitHub at: `https://github.com/justinscottlee/smile-rapture`

**Algorithm 1** Server-Side Main Algorithm

```
Load unstitched images
Create and connect to clients
Distribute images to clients
for each client do
    Wait to receive subset of partially stitched images
    Append received images to images list
end for
Demosaic the received partially stitched images all together
Output the maximally stitched images
```

**Algorithm 2** Client-Side Main Algorithm

```
while True do
    Wait to receive images from server
    Demosaic the received images
    Send demosaiced images back to server
end while
```

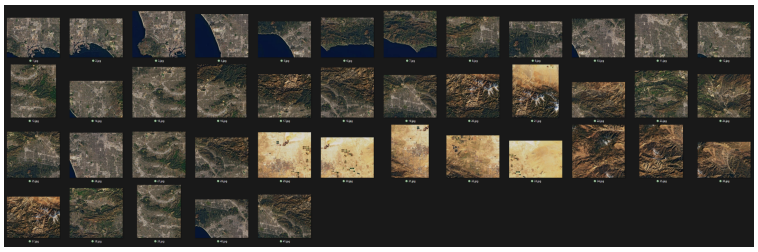

**Fig. 5 41 Satellite input images before stitching.**

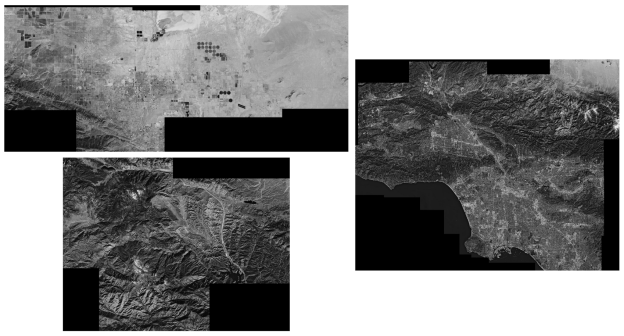

**Fig. 6 The three satellite imagery output mosaics.**

output and ensure the algorithm functions correctly, we intentionally separate the images into three disconnected sets that cannot be stitched further. Figure 6 displays the resulting stitched images, which match our expected three output mosaics.

This use case illustrates the utility of our platform for compute-centric applications and the ability for multiple nodes to collaboratively process user-uploaded experiments. Our platform can facilitate work on other computationally demanding workloads such as machine learning, autonomous algorithm developments, and large-scale data processing tasks. This flexibility makes it suitable for various research and academic applications where high-performance distributed computing is necessary.

### B. Case Two: Drone Collision Avoidance

UAS systems are a major research avenue in academia. In this use case, we create a proof-of-concept obstacle avoidance script that runs a state-of-the-art monocular depth pipeline to control the drone and avoid collisions with objects.

For demonstration purposes, a DJI Tello drone is used due to its easy-to-use Python API, cost-effectiveness, and minimal risk of damage thanks to its small size. The drone connects to a compute node, which acts as the master node for the UAS, issuing commands to the drone over WiFi. Our implementation employs the Metric3Dv2 [20] monocular depth technique for depth map computation. The depth map is computed by the compute node or any other node on the Kubernetes network, and the resulting depth data is used by the master node to generate and transmit movement commands to the drone over WiFi. Figure 7 shows an example output from the Metric3Dv2 pipeline in a room environment.


**Fig. 7 Depth map generated from Metric3Dv2 output with distance from the center pixel overlaid.**

The script performs takeoff and attempts to fly the drone straight until the collision avoidance algorithm detects an obstacle. Since this is only a proof-of-concept, the algorithm is relatively straightforward: it uses the depthmap to calculate the average distance to the group of 16×16 pixels in the middle of the image, and slows the forward velocity if this distance falls below a certain threshold. The drone then reorients itself until the avoidance criteria are no longer met.

This use case demonstrates how users can develop UAS applications and test them using RAPTURE. It also showcases how our platform allows users to run the latest algorithms and integrate them with hardware they may not have access to.

### C. Case Three: Rover Tracking

In the third use case, we demonstrate a rover node running on the platform and performing a real-time computer vision and control task. We use the rover control API built into the SMILE module to control the rover, and then apply OpenCV HSV thresholding to drive the rover toward a light source (see Figure 8).

In our implementation, the rover is driven by commands sent from an Arduino, which is directly connected to the rover via USB. The Arduino acts as the master node that controls that rover. A webcam is mounted on the rover, which is also directly connected to the same node via USB. These USB devices are accessible even in our containerized applications, ensuring control with minimal overhead. All peripherals are controllable via the SMILE module that users interface their code with. We provide an interface to control the rover via commands and an interface to open an OpenCV video stream from the webcam.

The computer vision pipeline consists of several steps. First, an image is retrieved from the webcam. Next, an HSV thresholded mask is created and smoothed using erosion and dilation. Contour detection is then applied to identify the largest feature in the mask, which becomes the target. The node rotates to center the feature and then drives toward it.

This use case demonstrates how our platform can support rover applications in addition to pure compute and UAS applications. We again include a computer-vision aspect in this demonstration, to show that our platform is well suited for tasks that involve computationally demanding workloads in real-time.

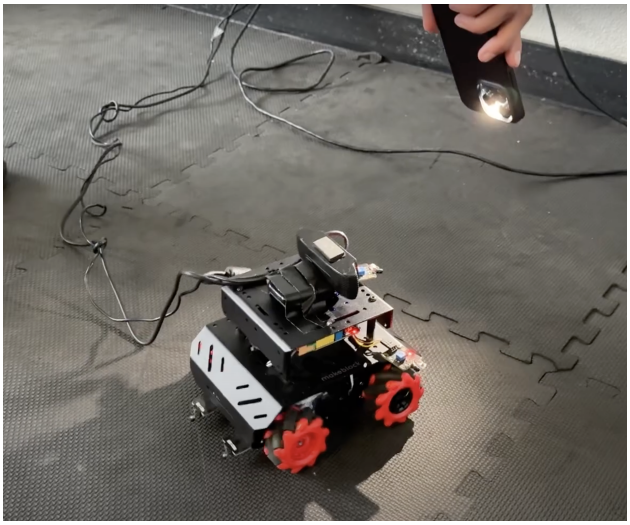

**Fig. 8 Rover tracking a light source and controlling toward it.**

## IV. System Overhead

Quantifying the performance and system overhead imposed by our platform is essential to understanding the impact of containerization and container allocation. In this section, we start by measuring the resource overhead that Kubernetes imposes on a node. Next, we assess the network overhead, and finally, we discuss the CPU penalty imposed on software running on our platform.

### A. Resource Overhead

We first assess our platform's resource overhead through benchmarking tasks running on our Kubernetes cluster. Our initial benchmark focuses on the CPU and memory resources utilized by the "k3s-agent" process. This process is crucial as it runs on worker nodes to join the cluster, manages communication with the master node, and executes workloads such as deploying and running containers.

| CPU Utilization | Memory Utilization |
|---|---|
| 1.9% | 162 MB |

**Table 1 Resource Utilization of the "k3s-agent" process on Raspberry Pi 4B.**

To measure the resource overhead, we utilize a Raspberry Pi 4B with 8GB of RAM as the compute node. The results are shown in Table 1. Additionally, we observed that the resources utilized by the "k3s-agent" process remain roughly constant, regardless of the number of applications.

### B. Networking Performance Overhead

We further assess the network performance overhead generated by our platform in latency and throughput under different conditions. We compare our approach with a native approach in which native Python code is run directly on the device (Raspberry Pi 4B 8GB). Each experiment is repeated 100 times to collect the latency and throughput results. The means and standard deviations are depicted in Figure 9. As we can see, latency, measured as the time to establish a connection, shows a significant 5.3× increase with a considerably higher standard deviation. Throughput, measured as the time to transmit a 10MB file, exhibits a 4.5% reduction.

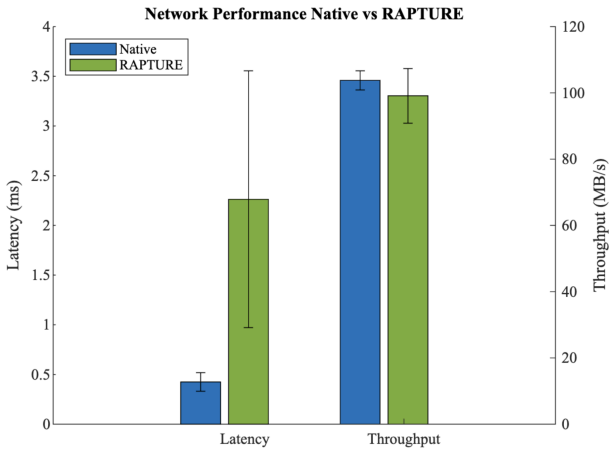


**Fig. 9 Comparison of network latency and throughput: Native vs. RAPTURE.**

The significantly poorer performance in latency can be attributed to Kubernetes' name resolution, which adds an additional software step compared to the direct IP connection used in the native implementation. This increased latency, along with the higher standard deviation, indicates more variability and potential delays in network communication within the Kubernetes-managed environment. Despite this, the higher mean latency (2.2ms) remains acceptable for most applications, including real-time applications like robotics, UAS, and computer vision systems. Furthermore, the advantages of containerization with our platform, such as ease of deployment and scalability, make the networking latency a minor issue.

Regarding throughput, we can observe a minor loss of performance. This can be attributed to the backend communication within Kubernetes itself, as well as the communication between nodes and the web portal.

### C. CPU Performance Overhead

Lastly, we assess the CPU performance overhead introduced by our platform. Similar to network overhead, we compare RAPTURE with a native implementation that runs Python scripts directly on the device (Raspberry Pi 4B). To measure CPU performance, we run a Fibonacci test that calculates a large Fibonacci number We then measure the CPU scores of the two methods, which are scalars inversely proportional to the computation time of the Fibonacci sequence. The means and standard deviations of their CPU scores, obtained over 100 runs, are shown in Figure 10. The

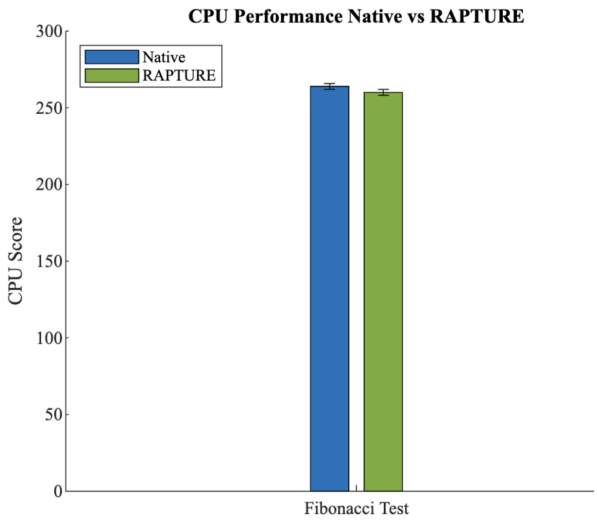


**Fig. 10 Comparison of CPU Performance: Native vs. RAPTURE.**

results reveal a minimal decrease of 1.5% in CPU performance, which is consistent with the earlier finding of 1.9% CPU utilization by the Kubernetes backend. Despite this slight decrease, the overall CPU performance remains largely unaffected, supporting the suitability of our platform for various computational tasks.

## V. Conclusion & Future Works

In this paper, we presented RAPTURE, an innovative containerized platform that democratizes access to a set of hardware testbeds, making research in UAS and robotics more accessible and effective. RAPTURE provides abstraction, result reporting, and support for various hardware platforms, including compute devices, ground rovers, and drones. The platform features a web portal that allows users to easily and remotely upload and run their experiments. To demonstrate the key features and potential of our platform, we explored three use cases: satellite image demosaicing, drone collision avoidance, and rover tracking. These use cases highlight the platform's capability to handle complex, distributed computing tasks across various domains and its potential to facilitate education and research in robotics, UAS, edge computing, and distributed systems, etc. We also evaluated the performance overhead introduced by our platform from various aspects, including resource utilization, latency, throughput, and CPU performance. The results show that the overhead is relatively minimal, with values of 1.9% for CPU utilization, 162 MB for memory usage, a 1.8ms increase in latency, a 4.5% reduction in throughput, and a 1.5% decrease in CPU performance. However, these slight performance trade-offs are outweighed by the benefits of containerization, such as ease of deployment and scalability, making our Kubernetes-based platform an effective solution for our target use cases.

### A. Future Works

In the future, we plan to enhance RAPTURE from the following key areas:

1) *Additional Programming Language Support:* Currently, RAPTURE supports only Python for user experiments. Future work could extend support to additional programming languages such as C/C++, Java, or Rust, which are widely used in robotics and UAS development. This expansion would broaden the user base and enable more computationally intensive experiments.
2) *Development of a Simulated Environment:* Simulation has become a crucial component in robotics, especially for developing complex multi-agent systems that are challenging and costly to test physically. To complement physical testbeds, we plan to incorporate a virtual simulator into RAPTURE, which accurately mirrors each hardware setup, enabling preliminary testing before deployment on actual drones. While many existing simulation environments, such as Gazebo [21], AirSim [22], and Webots [23], offer robust solutions, we propose leveraging one of these pre-developed software platforms. By creating middleware that allows experiments to run on our platform without code modifications, we can identify potential issues in a controlled and simulated environment, thus accelerating development and reducing the risk of errors before deploying code on real hardware.
3) *Integration with Cloud-Based Development Tools:* Cloud-based tools have greatly enhanced the developer experience by enabling greater collaboration and ease of use. This aligns with RAPTURE's primary goals of expanding access to UAS and robotics research and providing a platform that academia can easily use for iterative development in this field. Integrating RAPTURE with cloud-based development tools such as Jupyter Notebooks, Google Colab, or GitHub Codespaces would offer an even more streamlined way to conduct research. This integration would provide users with a seamless development environment, allowing them to write, test, and deploy code directly from their web browsers. Cloud-based environments also offer advantages such as easy collaboration, version control, and access to powerful computational resources, significantly streamlining the development and experimentation process.

## Acknowledgments

We would like to thank the National Science Foundation under Grants CAREER-2048266 and CCRI-2235157/2235158 /2235159/2235160 for the support of this work.

## References

[1] Islam, S., and Hu, X., "Real-time On-board Path Planning for UAS-based Wildfire Monitoring," *IEEE International Conference on Unmanned Aircraft Systems*, IEEE, Athens, Greece, 2021, pp. 527–535. https://doi.org/10.1109/ICUAS51884.2021.9476725.

[2] Grzybowski, J., Latos, K., and Czyba, R., "Low-cost autonomous UAV-based solutions to package delivery logistics," *Advanced,*

*Contemporary Control: Proceedings of KKA 2020—The 20th Polish Control Conference, Łódź, Poland, 2020*, Springer, 2020, pp. 500–507.

[3] Arriola-Valverde, S., Villagra-Mendoza, K., Méndez-Morales, M., Solórzano-Quintana, M., Gómez-Calderón, N., and Rimolo-Donadio, R., "Analysis of Crop Dynamics through Close-Range UAS Photogrammetry," *IEEE International Symposium on Circuits and Systems*, IEEE, Seville, Spain, 2020, pp. 1–5. https://doi.org/10.1109/ISCAS45731.2020.9181285.

[4] Niedzielski, T., Jurecka, M., Miziński, B., Pawul, W., and Motyl, T., "First Successful Rescue of a Lost Person Using the Human Detection System: A Case Study from Beskid Niski (SE Poland)," *Remote Sensing*, Vol. 13, No. 23, 2021. https://doi.org/10.3390/rs13234903.

[5] Su, J., Zhu, X., Li, S., and Chen, W.-H., "AI meets UAVs: A survey on AI empowered UAV perception systems for precision agriculture," *Neurocomputing*, Vol. 518, 2023, pp. 242–270.

[6] Huda, S. A., and Moh, S., "Survey on computation offloading in UAV-Enabled mobile edge computing," *Journal of Network and Computer Applications*, Vol. 201, 2022, p. 103341.

[7] Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., and Watteyne, T., "FIT IoT-LAB: A large scale open experimental IoT testbed," *IEEE World Form on Internet of Things*, IEEE, Milan, Italy, 2015, pp. 459–464. https://doi.org/10.1109/WF-IoT.2015.7389098.

[8] Wang, B., Xie, J., Li, S., Wan, Y., Gu, Y., Fu, S., and Lu, K., "Computing in the air: An open airborne computing platform," *IET Communications*, Vol. 14, No. 15, 2020, pp. 2410–2419. https://doi.org/10.1049/iet-com.2019.0515.

[9] Zhang, H., Wang, B., Wu, R., Xie, J., Wan, Y., Fu, S., and Lu, K., "Exploring Networked Airborne Computing: A Comprehensive Approach with Advanced Simulator and Hardware Testbed," *Unmanned Systems*, 2023.

[10] Wang, B., Xie, J., Ma, K., and Wan, Y., "UAV-based Networked Airborne Computing Simulator and Testbed Design and Implementation," *2023 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, 2023, pp. 479–486.

[11] Lu, K., Xie, J., Wan, Y., and Fu, S., "Toward uav-based airborne computing," *IEEE Wireless Communications*, Vol. 26, No. 6, 2019, pp. 172–179.

[12] Li, S., He, C., Liu, M., Wan, Y., Gu, Y., Xie, J., Fu, S., and Lu, K., "Design and implementation of aerial communication using directional antennas: learning control in unknown communication environments," *IET Control Theory & Applications*, Vol. 13, No. 17, 2019, pp. 2906–2916.

[13] Kubernetes Project Authors, "Kubernetes," , 2023. URL https://kubernetes.io/, accessed Dec 11, 2023.

[14] Lewis, G., Echeverría, S., Simanta, S., Bradshaw, B., and Root, J., "Tactical Cloudlets: Moving Cloud Computing to the Edge," *IEEE Military Communications Conference*, IEEE, IEEE, Baltimore, MD, USA, 2014, pp. 1440–1446. https://doi.org/10.1109/MILCOM.2014.238.

[15] Red Hat, "What is MicroShift?" , 2023. URL https://www.redhat.com/en/topics/edge-computing/microshift, accessed Dec 11, 2023.

[16] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A., "ROS: an open-source Robot Operating System," *IEEE International Conference on Robotics and Automation Workshop on Open Source Software*, 2009.

[17] Canonical, "MicroK8s," , 2023. URL https://microk8s.io, accessed Dec 11, 2023.

[18] K3s Project Authors, "K3s," , 2023. URL https://k3s.io, accessed Dec 11, 2023.

[19] Telenyk, S., Sopov, O., Zharikov, E., and Nowakowski, G., "A Comparison of Kubernetes and Kubernetes-Compatible Platforms," *IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, Vol. 1, IEEE, Cracow, Poland, 2021, pp. 313–317. https://doi.org/10.1109/IDAACS53288.2021.9660392.

[20] Hu, M., Yin, W., Zhang, C., Cai, Z., Long, X., Chen, H., Wang, K., Yu, G., Shen, C., and Shen, S., "Metric3D v2: A Versatile Monocular Geometric Foundation Model for Zero-shot Metric Depth and Surface Normal Estimation," , 2024.

[21] Koenig, N., and Howard, A., "Design and use paradigms for Gazebo, an open-source multi-robot simulator," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, Vol. 3, 2004, pp. 2149–2154 vol.3. https://doi.org/10.1109/IROS.2004.1389727.

[22] Shah, S., Dey, D., Lovett, C., and Kapoor, A., "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," , 2017.

[23] Michel, O., "Webots: Professional Mobile Robot Simulation," *Journal of Advanced Robotics Systems*, Vol. 1, No. 1, 2004, pp. 39–42. URL http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf.