PROWESS: An Open Testbed for Programmable Wireless Edge Systems

Jayson Boubin
Ohio State University
Columbus, Ohio, USA
boubin.2@osu.edu

Avishek Banerjee
Ohio State University
Columbus, Ohio, USA
banerjee.152@osu.edu

Jihoon Yun
Ohio State University
Columbus, Ohio, USA
yun.131@osu.edu

Haiyang Qi
Ohio State University
Columbus, Ohio, USA
qi.359@osu.edu

Yuting Fang
Ohio State University
Columbus, Ohio, USA
fang.564@osu.edu

Steve Chang
Ohio State University
Columbus, Ohio, USA
chang.136@osu.edu

Kannan Srinivasan
Ohio State University
Columbus, Ohio, USA
kannansrinivasan@cse.ohio-state.edu

Rajiv Ramnath
Ohio State University
Columbus, Ohio, USA
ramnath.6@osu.edu

Anish Arora
Ohio State University
Columbus, Ohio, USA
anish@cse.ohio-state.edu

ABSTRACT

Edge computing is a growing paradigm where compute resources are provisioned between data sources and the cloud to decrease compute latency from data transfer, lower costs, comply with security policies, and more. Edge systems are as varied as their applications, serving internet services, IoT, and emerging technologies. Due to the tight constraints experienced by many edge systems, research computing testbeds have become valuable tools for edge research and application benchmarking. Current testbed infrastructure, however, fails to properly emulate many important edge contexts leading to inaccurate benchmarking. Institutions with broad interests in edge computing can build testbeds, but prior work suggests that edge testbeds are often application or sensor specific. A general edge testbed should include access to many of the sensors, software, and accelerators on which edge systems rely, while slicing those resources to fit user-defined resource footprints. PROWESS is an edge testbed that answers this challenge. PROWESS provides access across an institution to sensors, compute resources, and software for testing constrained edge applications. PROWESS runs edge workloads as sets of containers with access to sensors and specialized hardware on an expandable cluster of light-weight edge nodes which leverage institutional networks to decrease implementation cost and provide wide access to sensors. We implemented a multi-node PROWESS deployment connected to sensors across Ohio State University's campus. Using three edge-native applications, we demonstrate that PROWESS is simple to configure, has a small resource footprint, scales gracefully, and minimally impacts institutional networks. We also show that PROWESS closely approximates native execution of edge workloads and facilitates experiments that other systems testbeds can not.

ACM Reference Format:
Jayson Boubin, Avishek Banerjee, Jihoon Yun, Haiyang Qi, Yuting Fang, Steve Chang, Kannan Srinivasan, Rajiv Ramnath, and Anish Arora. 2022. PROWESS: An Open Testbed for Programmable Wireless Edge Systems. In *Practice and Experience in Advanced Research Computing (PEARC '22), July 10–14, 2022, Boston, MA, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3491418.3530759

1 INTRODUCTION

Edge computing has arisen over the past decade as a solution to problems encountered by the Internet of Things (IoT). As IoT devices and sensors across the globe have accelerated in their production of data, the need for efficient near-sensor processing has increased [27]. While the availability and decreased cost of cloud computing provides data processing power to IoT devices, networks can struggle to transmit data between remote devices and the cloud. IoT sensors have limited compute capacity and tight power budgets, making them poor candidates for overprovisioning [2, 17]. Edge computing tackles this challenge by provisioning extra compute resources and creative networking solutions between IoT devices and the cloud. Edge devices store, cache, and process data that could otherwise saturate links to data centers. This helps facilitate many applications that generate large amounts of data, have critical latency bounds, or have security concerns that necessitate decentralization or data anonymization.

Edge systems are application-driven and encounter many disparate constraints including form factor, proximity to sensors, power budgets, compute capacity, and cost. The conflicting nature of these costs require implementers to consider many-way tradeoffs when designing software and selecting hardware. Testing these tradeoffs often requires investment in ineffective designs and hardware that does not meet requirements. Even if hardware

PEARC '22, July 10–14, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9161-0/22/07…$15.00
https://doi.org/10.1145/3491418.3530759

and software requirements are met, underlying characteristics of the deployment may impede optimal performance in ways that are not obvious without proper testing [7, 20]. For these reasons, custom and shared testbeds hold great value for edge researchers and practitioners.

The problem: Testbeds have long been used to design and benchmark applications before large-scale deployments [3, 11, 18]. Testbeds provide an environment in which hardware, software, models, and other artifacts of a deployment can be interchanged to determine optimal configurations. There are many edge testbeds [13, 14, 16, 19, 21–23, 30, 32] of varying complexity and generality. Most, however, focus on one application type, sensor, or attribute of an application. Furthermore, custom or highly focused testbeds often do not have open or extensible deployment platforms that can be translated across institutions or between large networks of users. Some broader computer systems testbeds, like Chameleon [18] and GENI [3], are highly general. They provide high degrees of user freedom, but are designed for the cloud, not the edge. These testbeds provide resource and network slicing, but allocate heavy-weight virtual machines or overprovisioned bare metal nodes that are far from sensors and not representative of edge resources. The problem is then that the principles of both general systems testbeds and bespoke edge testbeds have yet to be combined to facilitate general edge experimentation.

Edge testbed desiderata: Designing a testbed that is extensible, easy to deploy, and useful for general edge workloads comes with challenges. We identify five key principles that comprehensive edge testbeds should address: §1) *Rich Configurability*: As many aspects of the testbed as possible should be configurable by the user, including hardware, software, networking resources, and sensor access. §2) *Efficient Slicing*: Users should be able to request fine-grained slices of resources without encumbering their workload with unnecessary overhead. §3) *Bring Your Own Device (BYOD)*: The addition and allocation of sensors and nodes should be simple, meaning sensors and compute should easily integrate with the testbed, potentially across existing infrastructure. §4) *Infrastructure Leverage*: Edge testbeds should access already provisioned networks to ease deployment effort, leverage and integrate potentially large-scale and geographically disparate resources and sensors, and reduce cost. §5) *User and Infrastructure Security*: An edge testbed should assure that data transmission, storage, and access are secure, private, and controlled.

No extant testbed satisfies all of these principles. General systems testbeds like Chameleon and GENI both provide hardware configuration options, but both require users to allocate either entire bare-metal nodes which are rarely representative of edge contexts and require manual slicing, or sliced virtual machines which introduce unnecessary overhead. While GENI is decentralized, it requires significant administrative support to implement locally. Chameleon, on the other hand, is centralized, meaning it can not leverage local infrastructure for outside users. Edge testbeds also fail to meet our requirements. Most are designed for custom workloads [21, 22, 30] or small user groups [13, 19], providing little or no programability, configurability and slicing.

Our contribution: To address the problem described above, we present PROWESS: A slice-able, scalable, sensor-aware and open testbed architecture for general edge experimentation across institutions with provisioned networks that satisfies edge testbed desiderata §1-5. PROWESS, shown in Figure 1, is a collection of open source (upon publication) software artifacts and principles that bring edge computing experimentation to new institutions. PROWESS is totally configurable(§1), connecting sensors and compute resources important to edge experimentation to users via institutional access points(§4). PROWESS allows users to configure new compute nodes, sensors, and wireless networking solutions quickly, often in less than 100 lines of code. PROWESS runs user code in containers, allowing users to request compute slices(§2) and execute arbitrary code without incurring the overhead of virtualization or security infrastructure of bare-metal privileged access. Administrators can allocate sensors for public or restricted use through PROWESS' multicast interface, and users can bring their own sensors(§3) connected over the institutional network. By attaching to secure institutional networks and sandboxing user code in containers, prowess maintains a secure and private testbed for sensor workloads(§5).

We built the first PROWESS testbed implementation at Ohio State University (OSU), a large research-focused institution. Our PROWESS implementation contains 9 decentralized nodes provisioned on our OSU's wireless overlay network (OSU Wireless). OSU Wireless provides a rich set of over 20,000 access points which can be configured as PROWESS endpoints for sensor or hub configuration. Using this testbed, we built applications representing three different edge computing workloads: software defined radio localization, real-time audio stream analysis, and autonomous UAV image processing. Using these workloads, we demonstrate that: 1) PROWESS has a small footprint and is highly scalable, using at most 1.3 CPU cores on the core hub, 200MB of RAM, and 8Kbps of network bandwidth per provisioned hub. 2) PROWESS better represents the resource requirements of actual edge systems than other testbeds. Execution time of PROWESS workloads is within 1% of bare-metal execution time, where virtual machine execution takes between 29% and 50% longer. 3) PROWESS facilitates experiments that are not possible on other systems testbeds. We demonstrate using a simple SDR application that PROWESS avoids bandwidth constraints, network jitter, and latency to process SDR streams with 16-800X less overflows than cloud testbeds. 4) PROWESS hubs and sensors are simple to configure. We configure a PROWESS edge hub and two sensor nodes with only 54 lines of code.

Section II of this paper provides background on edge computing and systems testbeds. Section III presents the design of PROWESS. Section IV describes implementation details of PROWESS, and Section V covers our three test applications. Section VI details results comparing PROWESS to other testbed frameworks and an edge hub deployment case-study.

2 BACKGROUND AND MOTIVATION

As edge computing has matured, many testbeds have been developed for both general and specific use cases. Custom edge testbeds have been used to benchmark edge-specific algorithm and AI development [16, 32], edge networking techniques [21, 22], offloading [14], hardware paradigms [23] and diverse edge applications [7, 30]. Custom testbeds, however, lack the wide deployment,

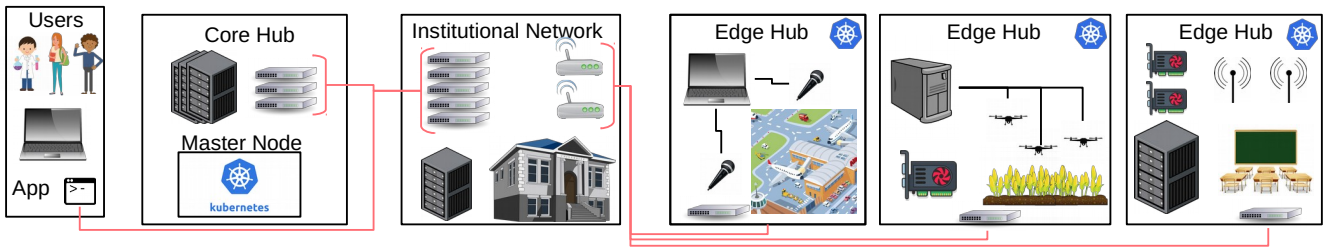

Figure 1: PROWESS places user apps on edge compute resources by leveraging existing institutional networks and common cloud-native technologies

rich sensor availability, re-programmability, and extensibility of major systems testbeds. Some edge testbeds, like USC's CCI IoT Testbed [13], are wide reaching and have considerable sensor access, but lack the programmability necessary for edge experimentation. Similarly, testbeds at Purdue University [19] are programmable, but lack the openness, orchestration, wide reach, and slicability that edge practitioners require of a general testbed.

In lieu of edge testbeds, large systems testbeds have been used for edge research. GENI [3] is a distributed and networked systems testbed with highly provisioned nodes across the United States. GENI users reserve compute resources as slices: isolated resources within the ecosystem on which users run experiments. GENI allows users fine-granined control over the network of a slice, allowing users to specify network parameters and protocols for experimentation. GENI resources have been used as edge systems to support a number of edge applications including connected autonomous vehicle (CAV) experimentation [15], AI-focused fog computing [24], and software-defined edge networking [25]. The Chameleon project, another popular systems testbed, has also been used for edge experimentation [18]. Chameleon allows users to configure virtual or bare-metal instances with accelerators and storage hierarchies to fit their needs. While both testbeds are useful for edge benchmarking, neither supports certain features required for edge applications. Both use either virtualization or bare metal instances only, making it difficult or impossible to precisely emulate the resource footprints of edge systems. Both also do not provide native access to sensor streams that edge systems rely upon. The Chameleon project has begun developing CHI@Edge, a container-driven Chameleon alternative to more closely approximate edge workloads and provide fine-grained slicing, but distance from edge sensors will still preclude many network-bound workloads from proper benchmaking.

In recent decades, the United States National Science Foundation has also funded major projects to build campus cyberinfrastructure that have resulted in strong developments. While not strictly a testbed, the Science DMZ [9] is a design pattern for high-performance networking specifically to facilitate data-intensive experimentation at research institutions. The Pacific Research Platform (PRP) is a cyber-infrastructure project that builds upon the Science DMZ model to deliver high-throughput networking to scientists across California. PRP and Science DMZ support numerous scientific projects in physics, computer science, biomedical science, and earth science [28], but focus mainly on inter-facility networking. These projects underscore the limitations of remote testbeds like Chameleon. Many scientific experiments require high-fidelity networking between compute nodes which can not be satisfied by public networks.

3 DESIGN

PROWESS involves both a compute architecture and software support for constrained edge experimentation. In this section, we describe the architecture, experiment lifecycle, sensor management, and user access.

3.1 Cluster Architecture

The PROWESS testbed is inherently a distributed system. As shown in Figure 2, PROWESS consists of at least one PROWESS core hub and a set of PROWESS edge hubs. Both the core hub and edge hubs are separate and potentially geographically disparate pools of compute resources connected by an institutional network. While edge and core hubs have key differences, both are fully configurable at the sensor, software, and hardware levels and inherently support experiment multi-tenancy. The core hub houses the majority of PROWESS software and system processing resources. The core hub also provides its own remaining resources for experiment scheduling. Edge hubs are additional members of the cluster that provide compute resources and may act as sensor endpoints.

Core Hub: The core hub runs all software that PROWESS requires for experiment scheduling using Kubernetes. PROWESS maintains a scheduling system described in Section 3.2 which runs entirely on the core hub. The PROWESS core hub hosts a sensor steam multicast module (described in Section 3.3), which pipes sensor streams to experiment containers. The PROWESS web portal also runs on the core hub. The PROWESS web portal (described in Section 3.4) is a secure Apache hosted web application which provides authenticated users access to PROWESS resources. Shown in section 6, core hub software has a small footprints, meaning a core hubs can be as light as a consumer laptop.

Edge Hub: Edge hubs function as augmented Kubernetes worker nodes positioned across the institutional network. Edge hubs can be anything from well-provisioned and centralized servers to far-flung and lightly provisioned embedded devices. Once provisioned, edge hubs are connected to the core hub by 1) joining the core hub's Kubernetes cluster, and 2) running a small set of custom PROWESS programs for sensor and network management.

Figure 2 shows three example edge hubs with with three different resource footprints based on our applications discussed

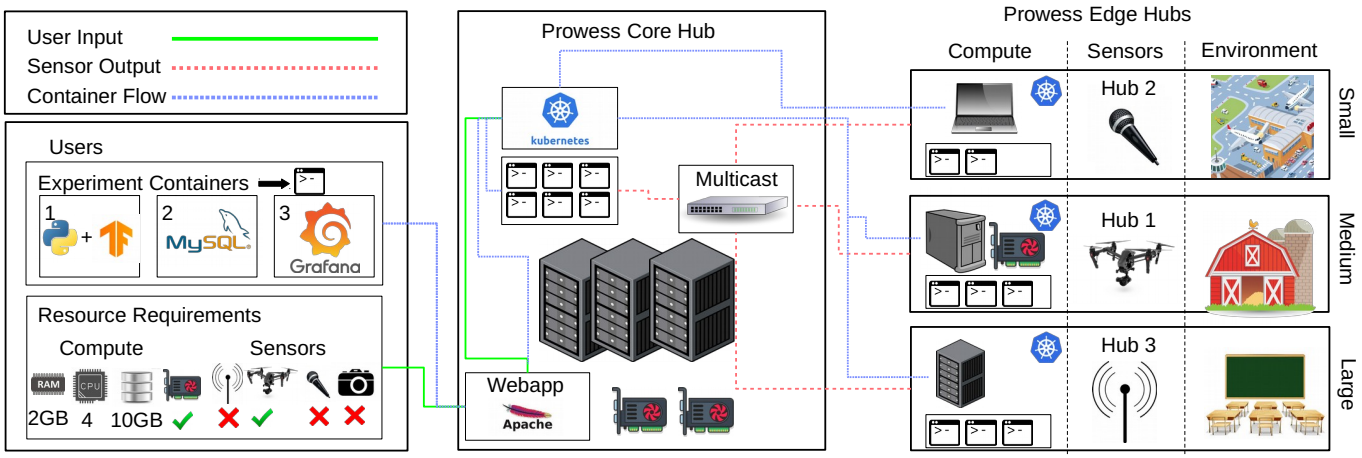

Figure 2: PROWESS design overview. Users define experiment containers and select required resources, which are scheduled across the PROWESS core hub and constituent edge hubs

in Section 5. Hub 1 uses only a consumer laptop to manage audio streams collected at an Airport, Hub 2 controls UAV using a desktop PC at a remote farm, and Hub 3 uses a server grade machine to control a software-defined radio provisioned in a classroom. All of these experiments have different sensors, requirements, and applications, but all can be easily provisioned through PROWESS.

3.2 Experiment Lifecycle

Figure 2 shows the PROWESS experiment lifecycle, which includes three steps: definition, scheduling, and execution. Definition begins with the user. Users define experiments as a set of Docker containers with optional shared volumes, and cluster specific domain names for data sharing and networking. PROWESS accepts containers as black boxes, meaning all software and configuration is defined by the user. PROWESS administrators may choose to provide baseline container images with system software or APIs to simplify user interaction with certain resources or sensors. Along with containers, PROWESS accepts resource requirements which include specifications for maximum allocations of RAM, CPU, storage, and network bandwidth. Users can also specify certain types of hardware, like storage disk or CPU type, GPU or TPU allocation, and more depending on availability throughout the cluster. Resource requirements include the sensor streams required for an experiment. Users identify the available sensors that each container requires for its experiment.

Once an experiment is defined, it can be scheduled in PROWESS. PROWESS provides a secure web interface for experiment resource definition, scheduling, and monitoring. Users provide PROWESS access to containers either through Dockerhub or a private institutional docker repository. Users also provide experiment date and runtime information through the web interface. Before an experiment is scheduled, PROWESS verifies that the requested resources will be available at the requested times, and that the user has the privilege to use certain resources or sensors. The PROWESS web portal provides anonymized utilization information for all connected hubs and nodes. For any given time, users are provided the remaining unreserved resources. Before a user can schedule an experiment, they must select the hub on which that experiment will run. If that hub has the resources and sensor access the user's experiment will be scheduled.

When an experiment is scheduled, it enters the PROWESS experiment queue. PROWESS jobs are scheduled using a combination of the base Kubernetes scheduler and the PROWESS job queue. The PROWESS job queue holds scheduled experiments that are waiting for their specified start time. When the job's scheduled start time is reached, the job is scheduled on its assigned hub through Kubernetes. PROWESS experiments hold their requested physical resources for the entirety of their execution. Each experiment is provided a directory, shared among all containers in the experiment, which will persist after the experiment concludes. When an experiment concludes, the persistent directory is compressed and made available to the user through the PROWESS web portal.

3.3 Sensors

Sensors are integral to the PROWESS experimentation model. Users often choose to deploy applications at the edge rather than in the cloud due to concerns over network fidelity or availability, data privacy, or insufficient throughput. Users should be able to connect to existing sensors provisioned specifically for PROWESS, or their own provisioned sensors. PROWESS provides software and leverages infrastructure for provisioned sensors through its sensor stream multicast module (SSMM), and by allowing users to securely connect sensors to containers through Aruba access points and across institutional networks.

The SSMM runs as system software on the core hub. PROWESS administrators can configure the SSMM to accept input streams from sensors positioned across the institutional network. So long as a sensor is connected to and authenticated on the institutional

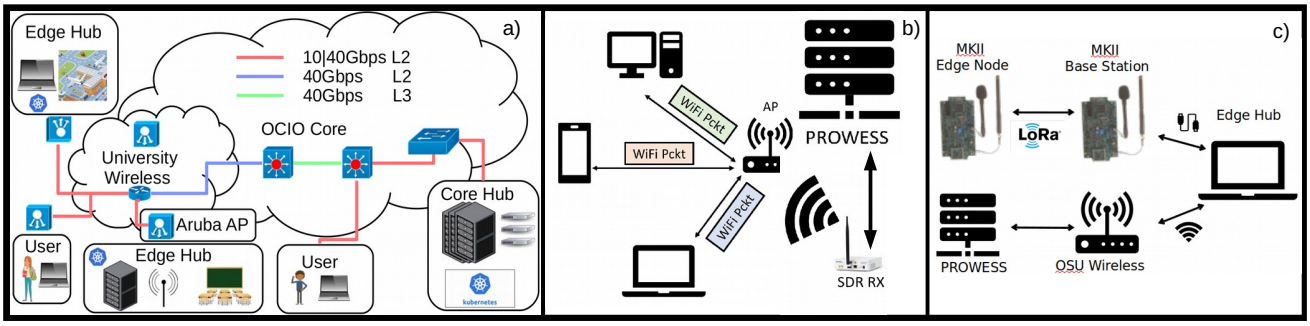

Figure 3: a) PROWESS Connects to OCIO Core but provisions edge hubs across OSU Wireless, b-c) Overview of our SDR and Acoustic classification application

network, it can be connected to PROWESS. The SSMM accepts UDP sensor streams from all configured sensors, and transmits sensor data to containers that request it via UDP multicast.

PROWESS allows users to bring their own sensors. PROWESS leverages new technology developed with Aruba Networks which allows for bi-directional communication between IoT gateways and edge devices directly through access points. Users of our PROWESS implementation can connect sensors via USB, BLE, or WiFi directly to one of over 20,000 access points at OSU and send data directly to PROWESS containers. PROWESS also provides every experimental container its own IP and DNS entry that is accessible through the Kubernetes cluster load-balancer on the PROWESS VPN.

3.4 Web Portal

PROWESS provides a web portal for users to schedule experiments, monitor workloads, and export data. The PROWESS web portal is built to integrate with common institutional single sign software. Our PROWESS prototype, for example, is integrated with OSU's Shibboleth [8] single sign-on system. PROWESS links authenticated users to their scheduled experiments and data using outputs from institutional tools, simplifying authentication and access control. Authenticated users can schedule experiments, view experiment and cluster status, and export data from prior experiments. To schedule experiments, users provide an experiment definition through a web form. Users are provided a calendar of anonymized experiments and their resource requirements so they can make informed decisions about when to schedule their jobs. Users can also monitor experiments status and outputs. PROWESS provides status information (e.g scheduled, running, complete, error) about all of a user's experiments within the system. When experiments conclude, users can export all data recorded in the experiments shared export volume as a zip file.

4 IMPLEMENTATION

We implemented a 9-node PROWESS prototype at OSU. Our PROWESS implementation consists of 1 core hub, 2 edge hubs, 4 embedded MKII audio nodes, and 2 USRP N210 SDRs. In this section, we describe the architecture, implementation, and configuration of this system.

Figure 3a shows the architecture of our PROWESS prototype. This prototype provisions one core hub and two edge hubs across OSU campus. The core hub sits behind the university's core network, OCIO Core. OCIO Core is comprised of layer 2 and 3 switches connecting hundreds of internal systems and services with a bandwidth of 10 and 40 Gbps. OSU Wireless is OSU's public wireless infrastructure which provides WiFi service to faculty, staff, students, and guestsin most University buildings. OSU Wireless connects users to 10-40Gbps layer 2 links through one of more than 20,000 Aruba wireless access points, each representing a potential PROWESS edge hub deployment, back to OCIO Core. PROWESS provisions the core hub behind OCIO Core and connects both edge hubs through OSU Wireless.

The PROWESS prototype core hub is located in Pomerene hall, a central building on OSU campus. The core hub is hosted on a Dell PowerEdge R815 server with an AMD Opteron 6128 CPU, 128GB of RAM, and 6 TB of disk space. Edge Hub 1, also located in Pomerene hall, is hosted on an HP ProLiant m710x server with a Xeon E3-1585L CPU, 64 GB of RAM, and 1TB of disk space. Edge Hub 2 is located at OSU Don Scott Airport, 6 miles northwest of campus. Edge Hub 2 is provisioned on an HP G6 laptop with a 2-core i5 7200u CPU, 4GB of RAM, and 500GB of disk space.

The core hub runs all PROWESS scheduling and management software, including the PROWESS web portal, the experiment scheduler, SSMM, and Kubernetes. The core hub also acts as a sensor endpoint for two MKII custom embedded audio sensors connected via Aruba APs. MKIIs use TDK InvenSense ICS-43432 microphones and embedded hardware to classify live audio, as described in Section 5. Edge hubs run Kubernetes worker software and all software required for the sensors they manage. Edge Hub 1 acts as a sensor endpoint for two USRP N210 software-defined radio. Edge Hub 2 acts as a sensor endpoint for two additional MKII nodes positioned at the airport. MKII Node 1 is positioned atop a National Ecological Observatory Network (NEON) tower at the north end of the airport. MKII Node 2 is positioned near the airport's main terminal and connected directly to Edge Hub 2 by USB. MKII Node 1 connects to MKII Node 2 using LoRa from 1.2km across the airfield.

5 APPLICATIONS

We implemented three edge-focused sensor-based applications on PROWESS to test and validate our design. Our applications include software-defined radio device identification, audio classification, and autonomous UAV feature extraction. In this section, we describe these three workloads in detail. In Section 6, we use them to test the performance of PROWESS.

SDR Wireless Device Identification: We developed an SDR application using PROWESS to count the number of wireless devices in a room. As shown in Fig. 3b, the SDR RX (receiver) behaves as a WiFi packet listener that sniffs into the nearby wireless environment. Wireless transmission between the end devices and Access Points (APs) are captured by our SDR application. These sniffed WiFi frames are then decoded and the MAC addresses of the transmitter (TX) and receiver (RX) are retrieved. To measure the number of wireless devices in a room we calculate the number of unique MAC addresses received during a given time slot. For the experimental setup of this application, we used a USRP N210 [12] software-defined radio. To sniff into WiFi 802.11n packets, the SDR needs to sample received signals at 20 MHz to match with the bandwidth of WiFi. PROWESS edge hub resources need to support this incoming high bandwidth stream and process it at a rate such that there are no buffer overflows, which result in packet loss and incomplete classification.

Real-time Audio Processing: Our second application classifies sound sources for remote monitoring of noise complaints. Our application operates in-situ in MKII nodes which compute the sound pressure level (SPL) of audio signals and classify noise pollution sources using machine learning [29]. MKII networks consist of several embedded edge nodes and one base station which wirelessly communicate using Long Range radio (LoRa). Edge nodes send classification results and SPL statistics to PROWESS-connected base station. Fig. 3c shows the stream of our acoustic application as deployed at the OSU airport.

Our acoustic application has been deployed at two different locations: the OSU airport (outdoor) and Pomerene lab (indoor). The airport deployment collects live audio data from the airfield to support learning of additional transportation noise classes for our noise pollution models. The lab deployment is focused on verifying the fitness of Aruba APs as MKII gateways, and providing the stream of MKII audio classification data to users through the PROWESS core hub. In this deployment, the MKII node is connected to an Aruba AP which sends results to the PROWESS core hub.

Autonomous UAV Feature Extraction: Autonomous unmanned aerial vehicles (UAV) are an emerging class of edge and embedded applications with tight power, compute, and latency budgets [4]. Autonomous UAV accomplish high-level goals without human intervention [6]. They perform tasks in domains like infrastructure inspection [1], agriculture [31, 33], search and rescue [26], and more. Autonomous UAV experience large and non-obvious performance impacts from naive hardware and software choices, making benchmarking critical. To test PROWESS' effectiveness for edge benchmarking, we implemented a feature extraction algorithm used for autonomous UAV pathfinding. When an autonomous UAV searches its environment to accomplish a goal, it must analyze the data it captures to determine 1) to what degree it has already accomplished its goal, and 2) where it should move next in pursuit of its goal. Conventionally, autonomous UAV use reinforcement learning for pathfinding. We implemented an algorithm from prior work [5] which uses feature extraction from UAV images to navigate environments using Q-learning.

6 RESULTS

In this section, we present findings from our PROWESS prototype and deployment. We found that PROWESS is easy to set up, uses few resources both on physical devices and across institutional networks, and closely approximates application performance on native hardware.

Figure 4 provides insight into the setup of PROWESS edge nodes and the resources they consume. Edge hubs are simple to deploy. Overall, only 54 lines of code (LOC) were required to configure an entire PROWESS edge hub. Figure 4 (a) shows the LOC required to set up our PROWESS edge hub at the OSU Airport which controlled two MKII audio sensors. All edge hub configurations are different, but baseline configuration requires little effort. Here, we required 13 LOC (installation) to install all PROWESS software with dependencies including Docker and Kubernetes, 8 LOC (edge) to connect the edge hub laptop to the core hub, 32 LOC (core) to configure the edge hub's YAML-based profile on the core hub, and 2 LOC (sensor) to configure our MKII nodes. PROWESS' reliance on industry-standard tools like Kubernetes and institutional networks makes deployment simple.

Figure 4 (b) shows the baseline resource utilization of our edge hub. The majority of the system is available for experiment scheduling. On average, testbed processes like Kubernetes and Docker take up on average 4.1% of one CPU, system processes consume 4% of one CPU, and the sensor ingestion application consumes 0.7% of one CPU. This leaves upwards of 95% of total compute resources (191.2% CPU of a dual-core machine) left over for scheduling.

Once configured, we found that experiments on PROWESS closely approximate bare-metal performance as shown in Figure 4 (c). We tested the performance of our UAV feature extraction workload on PROWESS, in a docker container, in a virtual machine, and on bare metal, all run on Edge Hub 1. Our docker container and virtual machine both used Ubuntu 18.04 with the same system software. Our virtual machine used the KVM hypervisor. For each workload, we allocated between 1 and 8 CPU cores through the PROWESS scheduling process and the docker and KVM command line interfaces. Bare-metal core allocation was performed by manually disabling CPU cores. Figure 4 shows that Bare-metal, Docker, and PROWESS have execution times within 1% of each other for all tested core allocations. KVM experienced a 29-50% performance penalty over bare metal, docker, and PROWESS, increasing as core count increases due contention for CPU resources through the KVM hypervisor. With the same underlying hardware, an 8-core KVM virtual machine implementation under-performs a 2-core PROWESS container or bare metal allocation.

PROWESS also facilitates the design and testing of applications that can not be performed with current well-known testbeds. Our SDR wireless device identification application has high bandwidth and low latency requirements that are easily met at the edge, but are difficult to meet with remotely provisioned resources. Figure 4 (d)

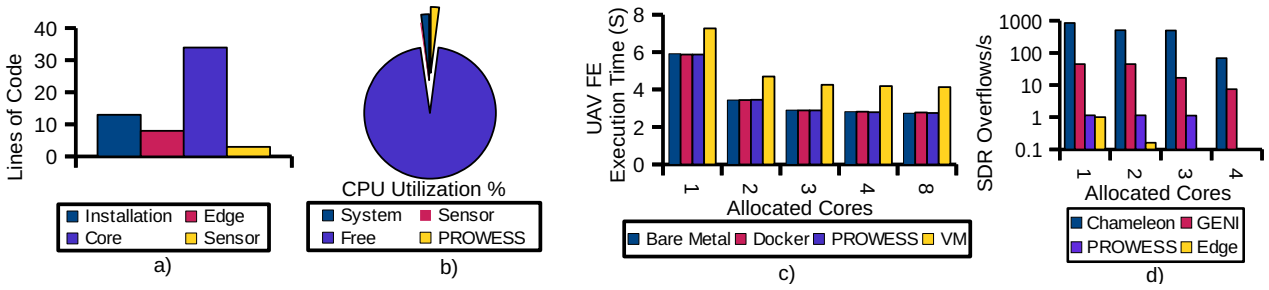

Figure 4: PROWESS closely approximates bare-metal performance in contrast to virtualization, and its proximity to sensors facilitates benchmarking that other testbeds can not.

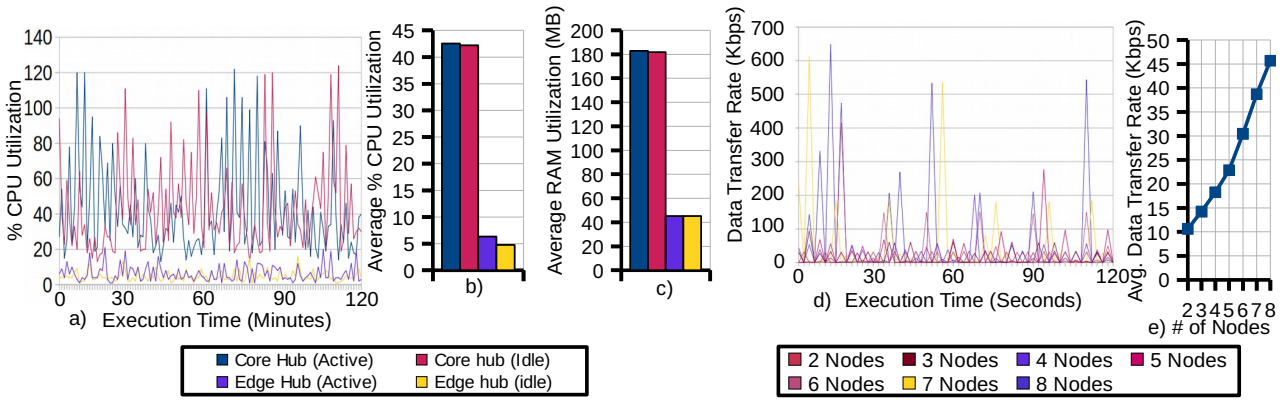

Figure 5: PROWESS nodes use resources parsimoniously and scale gracefully with respect to institutional networks

shows the performance of our SDR workload on GENI, Chameleon, PROWESS, and a native edge system. The native edge system was a Lenovo Thinkpad T470 laptop with an i7 7500u CPU and 24GB of RAM, running Ubuntu 18.04 Linux. The GENI experiment was run using our university's InstaGENI rack with an EMU-XEN 4-core virtual machine. Our Chameleon experiment used a bare-metal chameleon node situated at the Texas Advanced Computing Center.

Both Chameleon and GENI experience considerably more SDR overflows than PROWESS and edge hardware. Overflows occur when machines can't process incoming packet streams fast enough to keep buffers from overflowing. This can be mitigated by allocating more cores, or assuring consistent network connection and packet delivery. PROWESS experiences performance penalties compared to bare metal hardware as a result of the extra layer of Kubernetes networking. GENI experiences over an order of magnitude more overflows than PROWESS and edge hardware at all levels of core allocation. Similarly, chameleon experiences two to three orders of magnitude more overflows than PROWESS and edge hardware.

At the sample rate required to sniff WiFi packets, our SDR transmits data at 620Mb/s. Furthermore, edge hardware and PROWESS receive packets quickly, with a latency near 1ms. Our University GENI node receives packets in 2.5ms. Chameleon, positioned hundreds of miles away, has a latency of 37ms. Chameleon is unable to keep its buffers from regularly overflowing due to increased latency and jitter. SDR packets experience on average 228μs of jitter with Chameleon as compared to 11μs with native hardware and PROWESS, increasing the probability of buffer overflows. GENI is unable to keep up with packet processing due to the overhead of virtualization. Furthermore, both GENI and Chameleon do not have the bandwidth to receive all packets. GENI receives only 15% of all transmitted SDR packets, putting less pressure on buffers and limiting overflows. Overall, this type of workload requires a strong, consistent network connection and near bare-metal hardware performance to benchmark. Only PROWESS provides both of these features for users.

Figure 5 (a)-(c) shows a more detailed depiction of the footprint of both a PROWESS core hub and edge hub while idled and under load. To determine the resource footprint required to support PROWESS, we captured resource utilization information for every process spawned by PROWESS and its dependencies using perf [10]. Figure 5 (a) shows CPU utilization over time of both active and idle PROWESS edge and core hubs sampled once per minute over two hours. 'Active' CPU utilization was profiled only when experiments were being scheduled on that hub. CPU Performance is reported in percent CPU utilization, the aggregate percentage of 1 CPU that all

PROWESS processes are currently using. Figure 5 (a) shows that CPU utilization is highly variable for both edge and core hubs, but overall remains low, and is not considerably affected by activity. Our PROWESS core hub used between 13% and 131.9% CPU when idle and between 13% and 137.9% when active. Edge hubs require less resources, using between 1% and 17% CPU when idle and between 1% and 28% CPU when active. As reported by Figure 5 (b), Active and idle core hubs use on average only 42.5% and 42.2% of a single CPU core respectively. Active and idle edge hubs use 6.3% and 4.8% of a single core. PROWESS' memory footprint, shown in Figure 5 (c), follows a similar trend. Active and idle core hub processes reserve resident sets of on average 181.3MB and 186.5MB of RAM, while active and idle edge hubs reserve on average 45.19MB and 45.26MB. This footprint is quite small. A PROWESS core hub can be provisioned on a 2-core machine with 1GB of RAM and sufficient storage, while an edge hub requires even less resources. PROWESS hubs do not require high-end or specialty resources to be provisioned, meaning unused compute resources can be repurposed as testbed hubs. Both core and edge hubs could also feasibly be provisioned on Raspberry Pis or other similarly provisioned embedded systems.

Figure 5 (d) and (e) show how PROWESS scales as edge-hubs are added. Figure 5 (d) shows that most PROWESS traffic is periodic. Kubernetes heartbeat traffic consists the majority of the baseline network utilization of PROWESS. As PROWESS hubs are added, the average aggregate network utilization increases by around 5Kbps as seen in Figure 5, characterized by spikes in utilization of around 100Kbps as log and health information are transmitted. Smaller spikes were observed when the Kubernetes command-line took Kubectl is invoked for experiment scheduling. The overall network footprint of PROWESS is small compared to the institutional network on which it is provisioned as shown in Figure 3 (a). At peak, an 8-hub PROWESS instance requires 648Kbps bandwidth to support baseline PROWESS inter-hub communication on our 10-40Gbps institutional network, giving PROWESS room to grow.

7 CONCLUSION

PROWESS is a testbed built specifically for edge experimentation. PROWESS uses cloud-native technologies and existing infrastructure to schedule and manage edge and sensor-based experiments across universities and research institutes. PROWESS has a small resource footprint, allowing experimentation on sparsely provisioned consumer and embedded devices. PROWESS has limited network impact and scales gracefully, allowing researchers to deploy PROWESS nodes widely with little concern of network pressure. We implemented and tested PROWESS at our University, demonstrating that PROWESS closely approximates native hardware execution for edge workloads, facilitates edge experiments that other testbeds can not, and allows edge hub deployment in less than 100 lines of code.

ACKNOWLEDGMENTS

This work was funded in part by an NSF CC* Integration Award (2018912) and an NSF Graduate Research Fellowship (DGE-1343012).

REFERENCES

[1] Abdulla Al-Kaff, Francisco Miguel Moreno, Luis Javier San José, Fernando García, David Martín, Arturo de la Escalera, Alberto Nieva, and José Luis Meana Garcéa. 2017. VBII-UAV: Vision-based infrastructure inspection-UAV. In *World Conference on Information Systems and Technologies*. Springer, 221–231.

[2] Anish Arora, Prabal Dutta, Sandip Bapat, Vinod Kulathumani, Hongwei Zhang, Vinayak Naik, Vineet Mittal, Hui Cao, Murat Demirbas, Mohamed Gouda, et al. 2004. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks* 46, 5 (2004), 605–634.

[3] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. 2014. GENI: A federated testbed for innovative network experiments. *Computer Networks* 61 (2014), 5–23.

[4] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Reddi. 2018. MAVBench: Micro Aerial Vehicle Benchmarking. In *MICRO*.

[5] Jayson Boubin, Codi Burley, Peida Han, Bowen Li, Barry Porter, and Christopher Stewart. 2021. Programming and Deployment of Autonomous Swarms using Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2105.10605* (2021).

[6] Jayson Boubin, John Chumley, Christopher Stewart, and Sami Khanal. 2019. Autonomic Computing Challenges in Fully Autonomous Precision Agriculture. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE.

[7] Jayson G Boubin, Naveen TR Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. 2019. Managing edge resources for fully autonomous aerial systems. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. ACM, 74–87.

[8] Scott Cantor and T Scavo. 2005. Shibboleth architecture. *Protocols and Profiles* 10 (2005), 16.

[9] Eli Dart, Lauren Rotman, Brian Tierney, Mary Hester, and Jason Zurawski. 2014. The science dmz: A network design pattern for data-intensive science. *Scientific Programming* 22, 2 (2014), 173–185.

[10] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[11] Emre Ertin, Anish Arora, Rajiv Ramnath, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, Hui Cao, and Mikhail Nesterenko. 2006. Kansei: A testbed for sensing at scale. In *Proceedings of the 5th international conference on Information processing in sensor networks*. 399–406.

[12] Matt Ettus. 2005. Usrp users and developers guide. *www. olifantasia. com/gnuradio/usrp/files/usrp_guide. pdf* (2005).

[13] USC Center for Cyber-Physical Systems and the Internet of Things. 2021. A Campus-wide internet-of-Things Testbed. http://cci.usc.edu/index.php/cci-iot-testbed/.

[14] Hend Gedawy, Sannan Tariq, Abderrahmen Mtibaa, and Khaled Harras. 2016. Cumulus: A distributed and flexible computing testbed for edge cloud computational offloading. In *2016 Cloudification of the Internet of Things (CIoT)*. IEEE, 1–6.

[15] Abhimanyu Gosain, Mark Berman, Marshall Brinn, Thomas Mitchell, Chuan Li, Yuehua Wang, Hai Jin, Jing Hua, and Hongwei Zhang. 2016. Enabling campus edge computing using geni racks and mobile resources. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 41–50.

[16] Tianshu Hao, Yunyou Huang, Xu Wen, Wanling Gao, Fan Zhang, Chen Zheng, Lei Wang, Hainan Ye, Kai Hwang, Zujie Ren, et al. 2018. Edge AIBench: towards comprehensive end-to-end edge computing benchmarking. In *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 23–30.

[17] Soumil Heble, Ajay Kumar, KV V Durga Prasad, Soumya Samirana, Pachamuthu Rajalakshmi, and Uday B Desai. 2018. A low power IoT network for smart agriculture. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 609–614.

[18] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. 2020. Lessons learned from the chameleon testbed. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 219–233.

[19] Purdue Dependable Computing Systems Laboratory. 2021. Facilities and Equipment at DCSL: Testbeds. https://engineering.purdue.edu/dcsl/about/.

[20] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *ASPLOS*.

[21] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. 2019. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2287–2295.

[22] Raul Muñoz, Laia Nadal, Ramon Casellas, Michela Svaluto Moreolo, Ricard Vilalta, Josep Maria Fàbrega, Ricardo Martínez, Arturo Mayoral, and Fco Javier Vílchez. 2017. The ADRENALINE testbed: An SDN/NFV packet/optical transport network and edge/core cloud platform for end-to-end 5G and IoT services. In *2017 European Conference on Networks and Communications (EuCNC)*. IEEE, 1–5.

[23] Jianli Pan, Lin Ma, Ravishankar Ravindran, and Peyman TalebiFard. 2016. HomeCloud: An edge cloud framework and testbed for new application delivery. In *2016 23rd International Conference on Telecommunications (ICT)*. IEEE, 1–6.

[24] Jon Patman, Meshal Alfarhood, Soliman Islam, Mauro Lemus, Prasad Calyam, and Kannappan Palaniappan. 2018. Predictive analytics for fog computing using machine learning and GENI. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 790–795.

[25] Alessio Sacco, Flavio Esposito, and Guido Marchetto. 2020. A federated learning approach to routing in challenged sdn-enabled edge networks. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 150–154.

[26] Jürgen Scherer, Saeed Yahyanejad, Samira Hayat, Evsen Yanmaz, Torsten Andre, Asif Khan, Vladimir Vukadinovic, Christian Bettstetter, Hermann Hellwagner, and Bernhard Rinner. 2015. An autonomous multi-UAV system for search and rescue. In *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*. 33–38.

[27] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[28] Larry Smarr, Camille Crittenden, Thomas DeFanti, John Graham, Dmitry Mishin, Richard Moore, Philip Papadopoulos, and Frank Würthwein. 2018. The pacific research platform: Making high-speed networking a reality for the scientist. In *Proceedings of the Practice and Experience on Advanced Research Computing*. 1–8.

[29] Sangeeta Srivastava, Dhrubojyoti Roy, Mark Cartwright, Juan P Bello, and Anish Arora. 2021. Specialized Embedding Approximation for Edge Intelligence: A Case Study in Urban Sound Classification. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 8378–8382.

[30] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Ranveer Chandra, Anish Kapoor, Sudipta Sinha, Madhusudhan Sudarshan, and Sean Stratman. 2017. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *NSDI*.

[31] Ming-Der Yang, Jayson G Boubin, Hui Ping Tsai, Hsin-Hung Tseng, Yu-Chun Hsu, and Christopher C Stewart. 2020. Adaptive autonomous UAV scouting for rice lodging assessment using edge computing with deep learning EDANet. *Computers and Electronics in Agriculture* 179 (2020), 105817.

[32] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. 2019. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1270–1278.

[33] Zichen Zhang, Jayson Boubin, Christopher Stewart, and Sami Khanal. 2020. Whole-Field Reinforcement Learning: A Fully Autonomous Aerial Scouting Method for Precision Agriculture. *Sensors* 20, 22 (2020), 6585.