Junkyard Computing: Repurposing Discarded Smartphones for Serverless Applications

AJAY RAMESH RANGANATHAN, University of California, San Diego, USA
RISAB SANKAR, University of California, San Diego, USA
GRANT CHENG, University of California, San Diego, USA
AYAN GAUR, University of California, San Diego, USA
ARUNAN THIVIYANATHAN, University of California, San Diego, USA
JEFFREY LEE, University of California, San Diego, USA

The global demand for computing resources continues to grow rapidly, while billions of smartphones are discarded annually despite possessing high-performance CPUs and GPUs. This, however, presents a unique opportunity: repurposing discarded mobile devices as compute nodes for distributed systems. The Junkyard Project explores this potential by transforming discarded smartphones into a datacentre. We develop a lightweight distributed computing stack using containerized workloads managed via Kubernetes and Docker. Our system supports both serverless workloads and high-performance computing applications. We evaluate our system through two use cases: (1) Green Grader, an automated academic grading system and (2) FishSense, a computer vision workload optimized for distributed inference. Our results prove the feasibility of running distributed serverless application on a phone cluster. For instance, we find that even a small cluster of 4 phones can process 33 grading tasks in a minute.

Additional Key Words and Phrases: Carbon Footprint, Smartphone Cluster, Serverless Computing, Distributed Systems, Container Orchestration

ACM Reference Format:
Ajay Ramesh Ranganathan, Risab Sankar, Grant Cheng, Ayan Gaur, Arunan Thiviyanathan, and Jeffrey Lee. 2018. Junkyard Computing: Repurposing Discarded Smartphones for Serverless Applications. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (August 2018), 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The rapid growth of the smartphone market and the associated frequent upgrades contribute significantly to the escalating problem of electronic waste. In 2022 alone, an estimated 5.3 billion mobile phones were projected to be discarded, forming a significant portion of the estimated 62 million tonnes of e-waste generated worldwide. At the time of their disposal, many of these devices remain functionally intact and contain advanced computing hardware. This disconnect between capability and lifecycle presents an opportunity for sustainable innovation.

The Junkyard Project addresses this issue by demonstrating how discarded smartphones can be transformed into a data centre that is capable of distributed computing and running serverless applications. Serverless

Authors' Contact Information: Ajay Ramesh Ranganathan, University of California, San Diego, San Diego, USA, arameshranganathan@ucsd.edu; Risab Sankar, University of California, San Diego, San Diego, USA, rsankar@ucsd.edu; Grant Cheng, University of California, San Diego, San Diego, USA; Ayan Gaur, University of California, San Diego, San Diego, USA; Arunan Thiviyanathan, University of California, San Diego, San Diego, USA; Jeffrey Lee, University of California, San Diego, San Diego, USA.

ACM 2476-1249/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

applications represent a computing execution model where the provider dynamically manages the allocation and provisioning of servers. The name ‘serverless" is a bit of a misnomer; it doesn’t mean there are no servers, but rather that developers don’t have to worry about managing, provisioning, or scaling them. The underlying server infrastructure is entirely abstracted away by the provider.

Our work focuses on constructing a data center using repurposed Google Pixel Fold smartphones, interconnected via Ethernet. This cluster forms a wired Local Area Network (LAN), receiving requests from the outside world and distributing their workloads across the devices.

To showcase the practicality and scalability of this system, we support two applications representative of real-world computational demands: Green Grader, an automated academic grading tool, and FishSense, a computer vision pipeline that leverages the cluster for distributed processing. These applications were chosen to demonstrate the cluster’s capability to support both high-throughput task parallelism and distributed workloads. Our work demonstrates the viability of repurposing discarded smartphones by developing and standardizing the operating system into a distributed computing environment using tools like Kubernetes and Docker. We offer an energy efficient, low-cost, and sustainable alternative for general-purpose computing tasks such as automated academic grading and machine learning inference, while reducing electronic waste and carbon footprint.

1.1 GreenGrader

Green Grader is a distributed automated grading system designed to offload the evaluation of programming assignments from centralized cloud platforms to our repurposed phone cluster. Each student submission is treated as an independent Kubernetes Job, enabling parallel execution across multiple nodes. This architecture supports high-throughput grading and reduces overall turnaround time, particularly in large courses with hundreds of submissions.

A Job server was developed to receive requests for creating tasks (jobs) in the phone cluster. This server is capable of concurrently processing requests and monitoring the scheduled jobs.

Our results show the effectiveness of the system in processing jobs concurrently and its feasibility for running automated grading applications. By exploiting the inherent concurrency and scalability of the cluster, the Green Grader presents a viable, energy-efficient alternative to conventional grading infrastructures.

1.2 FishSense

FishSense is a project at UCSD that monitors fish species to keep track of their population growth and general health. It mainly involves computer vision algorithms that execute on batches of inputs. It is therefore a good candidate for distributed computing.

We use Ray with Kubernetes to run image segmentation workloads on the phone cluster. Again, we show the ability of the cluster to process tasks concurrently with reasonable latency.

2 Related Work

The idea of smartphone repurposing and distributed computing as a means to support environmental sustainability has been increasingly explored in recent years due to rapidly increasing e-waste and growing computational demands. This section reviews relevant work across three primary areas: distributed mobile computing architectures, smartphone repurposing methods, and sustainable computing approaches.

2.1 Distributed Mobile Computing

Early work in distributed computing focused on leveraging idle devices for computational tasks. CWC (Computing with Charging) demonstrated the viability of distributed mobile computing by implementing a scheduling algorithm that minimized the makespan for distributed tasks, achieving 1.6x faster completion times compared to

similar approaches [1]. Despite establishing the feasibility of smartphone clustering, this work primarily focused on temporary idle devices as opposed to repurposed hardware.

Building on this work, DroidCluster implemented a proof-of-concept 6-node Android cluster running LINPACK benchmarks via MPI, demonstrating that smartphones possess computational power comparable to high-end workstations [3]. To address limitations of infrastructure cloud dependencies, Sanches proposed a framework that processes batch and streaming data with mobile device clouds, implementing a "move computation to data" strategy that reduces inter-device data exchange [4].

Further progress towards heterogeneous computing environments have also been made as exemplified by Molecule: this device introduced the first serverless computing system designed to work across different types of hardware rather than identical machines, achieving higher function density and better performance [13].

2.2 Smartphone Repurposing

The desire for smartphone repurposing has been supported by research showing that 1.5 billion smartphones are sold annually while most are decommissioned within two years despite remaining functional [2]. Suckling and Lee's life cycle analysis revealed that manufacturing dominates greenhouse gas emissions for smartphones [9], validating the environmental benefits of device lifetime extension.

The most directly relevant work to ours comes from Switzer et al., who developed a cloudlet using Pixel 3A phones running complete microservice-based applications [5]. They introduced a Computational Carbon Intensity (CCI) as a metric that balances service of older devices against runtime improvements of newer machines. The smartphone clusters were 9.8x to 18.9x more carbon efficient than equivalent AWS EC2 instances across different workloads.

To expand on this progress, Ward and Gittens proposed two frameworks to evaluate smartphone suitability for reuse: the General Repurposing Status Categorizing Model (GRSCM) for acquisition and classification of retired phones and the General Attribute/Sensor Identification Model (GASIM) for cataloging available sensors and capabilities in phones [6]. These approaches address the challenge that retired devices exist in various states of functionality. Norbisrath et al. further developed practical deployment methodologies for converting any Android smartphone after 2014 into IoT edge gateways using tools such as F-Droid and Termu [7].

2.3 Sustainable Computing

At the center of smartphone repurposing and distributed computing lies the goal of implementing energy-efficient distributed systems. Renée demonstrated that function-as-a-service (FaaS) capabilities could be implemented using discarded smartphones, with the local cluster achieving faster response times than commercial FaaS providers [8]. The system's architecture model with central management for task distribution and power control provides the model by which we expand on in our paper. Energy efficiency consideration was further explored by Patros et al. who examined energy consumption patterns in serverless computing functions and proposed various energy reduction strategies such as a pay-per-use economic model [14].

Along with the growth of energy-efficient distributed systems has been the emergence of container-based edge computing. Von Leon et al. explored using containers to build edge computing systems on small, distributed clusters [12]. This work evinced that containerized applications can successfully run on distributed clusters of small, low-power devices.

While there has been significant progress in demonstrating technical feasibility of smartphone repurposing and distributed computing, prior research has focused on temporary idle devices or simple proof-of-concept implementations. The Junkyard project addresses these limitations by developing a distributed computing platform designed for a repurposed phone cluster. The deployment of complex applications like automated grading systems and computer vision workloads offers a new contribution to the field as well.

Fig. 1. Phone cluster

3 Setup

The phone cluster is a network of 16 phones capable of communication with each other and the outside world. The phones' ability to function as a cluster is dependent on several factors such as the Operating System, network and the cluster management software (in our case Kubernetes). The cluster is shown in Figure 1.

3.1 Phone Specifications

The Google Pixel Fold is a foldable phone powered by Google's Tensor G2 chipset and eight ARM Cortex CPUs. The device has 12GB of RAM and up to 512GB of internal storage, and an ARM Mali G710 GPU.

3.2 Operating System

The phones run Postmarket OS which is based on Alpine Linux. The kernel configuration was updated to support network configs required by Kubernetes. These configs are mainly related to NETFILTER and NETLINK. For example, CONFIG_IP_ROUTE_CLASSID, CONFIG_NETFILTER_FAMILY_BRIDGE, CONFIG_NETFILTER_NETLINK_OSF, etc.

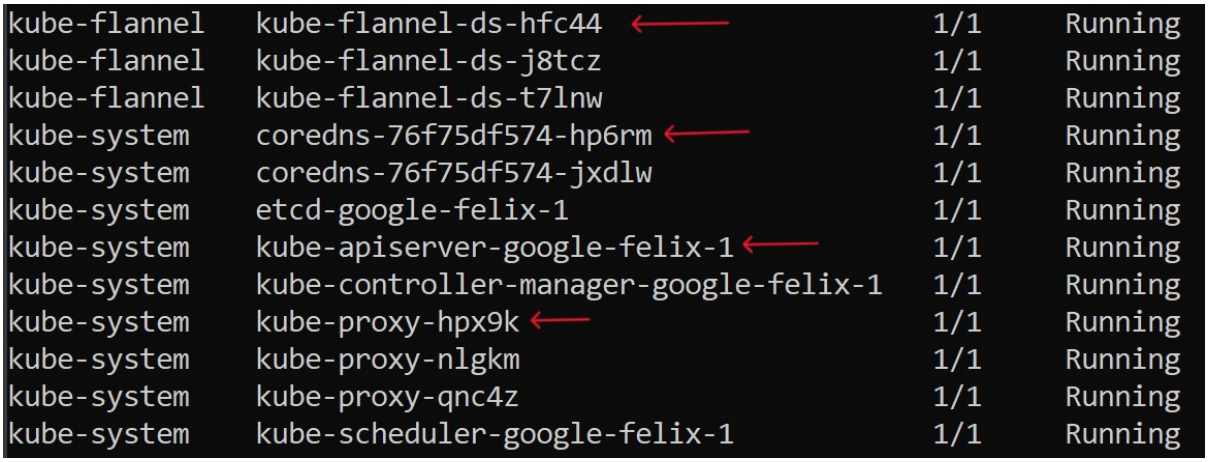

Fig. 2. Network related pods in the cluster. A 'Running' state indicates that the network is healthy.

3.3 Network

The cluster is connected to the lab's workstation via ethernet. It forms a local area network making it easy for the phones to access one another. The connectivity across the cluster was tested using Kubernetes default network pods, which are designed to run basic tests to check the health of the cluster. For example, Figure 2 shows the status of all pods in the cluster. The pods related to the state of the network are marked in red. If all in-built tests pass, the pods' status would be 'Running' and a 'ready' state (1/1). This indicates that the network is successfully set up and running.

3.4 Control plane and Worker Nodes

One of the phones is designated as the master, called the control plane in Kubernetes terminology. The control plane contains the necessary components to manage a cluster such as an API server to facilitate communication, a network manager to maintain network configuration, and a scheduler to schedule jobs. Worker nodes can join the cluster by authenticating with the control plane.

3.5 Cluster Management

Application deployment is facilitated by containerization. Applications are packaged into Docker containers that encapsulate language runtimes (e.g., Python, Java, OpenCL) and system dependencies, allowing deployment across heterogeneous devices. We utilize containerd and nerdctl for lightweight container runtime management and employ Kubernetes as the orchestration layer responsible for job scheduling, resource allocation, and fault recovery.

4 GreenGrader

This section discusses the design of the *GreenGrader* in detail. We start by describing the design at a high-level and then explain each component in detail.

4.1 System Design

The GreenGrader system consists of three main components - the phone cluster, the cluster management system, i.e., Kubernetes, and an online platform to manage assignment grading and submissions e.g., Gradescope. An

overview of the system is illustrated in Figure 3. The phone cluster consists of a master node, called the control plane, and worker nodes. A server that accepts requests from the outside world runs on one of the worker nodes. We call this the Job server.

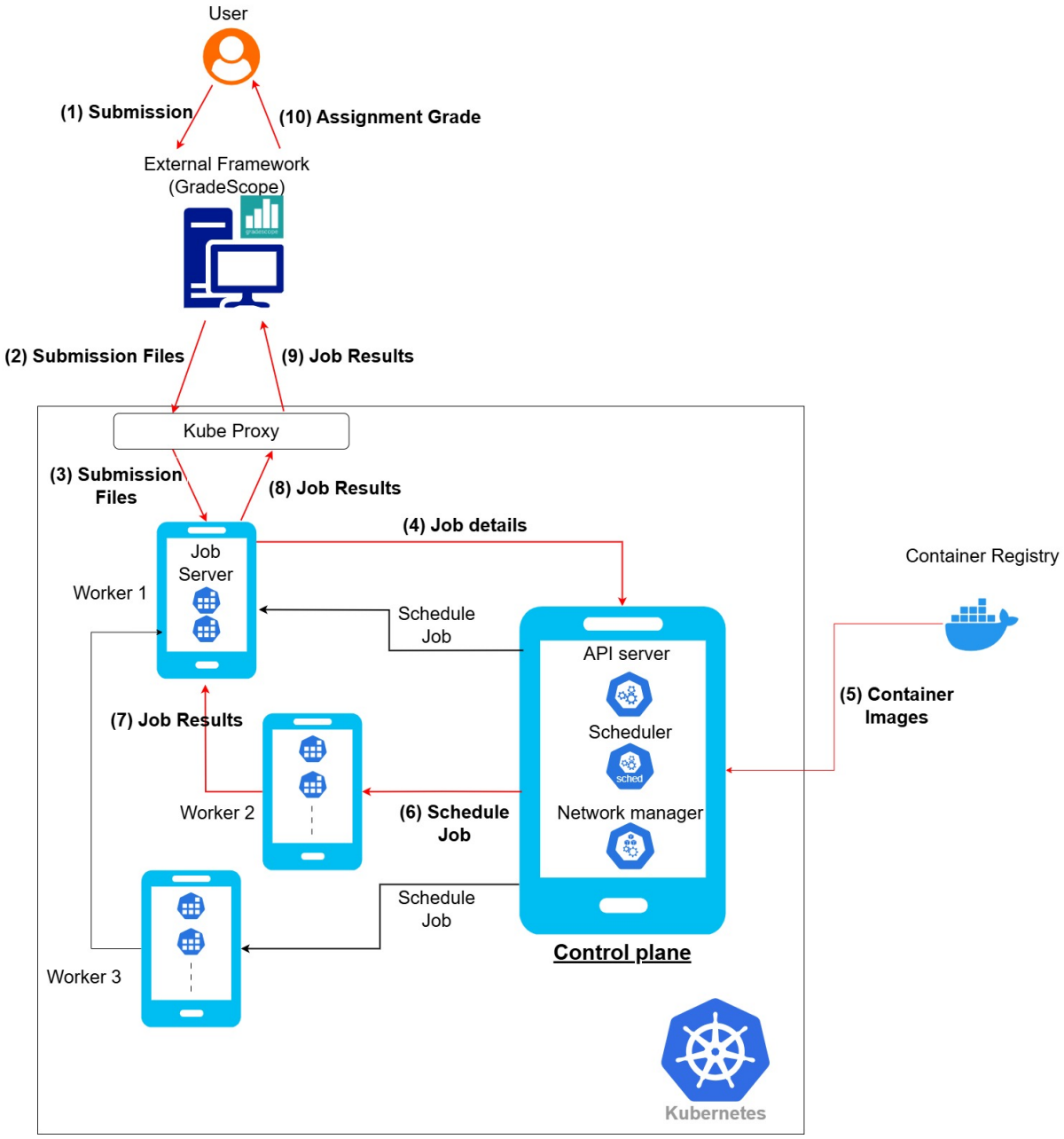

Fig. 3. System Design of the GreenGrader

Overall, the workflow of the system can be summarized in the following steps

(1) User submission to GradeScope: The user submits assignment files to GradeScope.
(2) Packaging of submission and transfer to the cluster: The submission files are packaged in the required format and sent to the Job server
(3) The transmitted data goes through the Kube proxy which is a component in Kubernetes that manages network rules on each node, ensuring efficient communication between services and pods. It operates as a network proxy, reflecting services as defined in the Kubernetes API on each node.
(4) The Jobserver processes the received data, and requests the control plane to create a new job.
(5) A container that contains the environment required to run the job is pulled from a remote repository.
(6) The control plane schedules a new job in a worker node in the cluster.
(7) When the job is complete, the worker node sends the results to the Job server.
(8) The Job server sends the results to GradeScope.
(9) The Kube proxy is involved again.
(10) Gradescope sends the results to the user. This is mainly in the form of visual output in GradeScope's webpage.

An example of users submitting assignments in GradeScope is shown in Figure 4. The autograder is triggered and runs on the phone cluster immediately after a submission as shown in Figure 5. After a while, the results are displayed in GradeScope as seen in Figure 6. Each of the components of the system is described in detail in following sections.

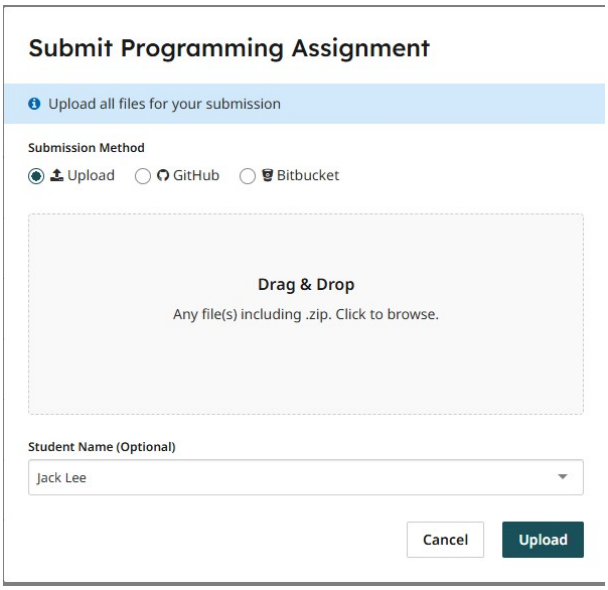

Fig. 4. Submission to GradeScope

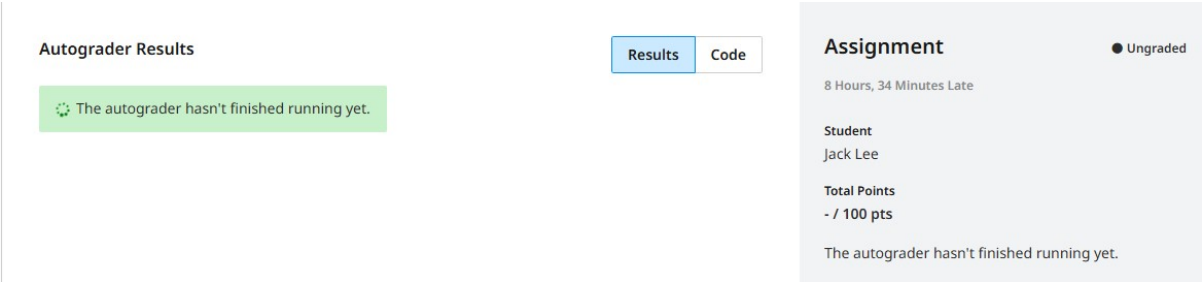

Fig. 5. Student waits for autograder

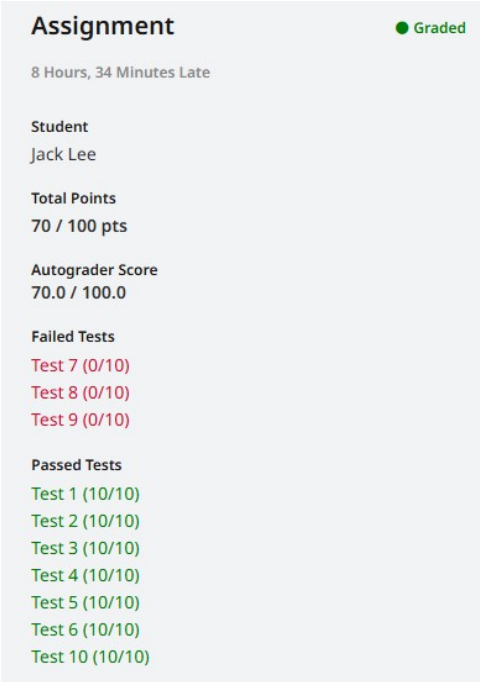

Fig. 6. Results of autograding displayed to the student in GradeScope

4.1.1 GradeScope. The *GreenGrader* system is integrated with Gradescope, a widely adopted online platform for managing and grading student-submitted assignments in higher education, including universities such as UC San Diego. A key feature of Gradescope is its autograder functionality, which enables student code to be automatically evaluated against an instructor-defined test script. Gradescope executes these scripts by provisioning an Ubuntu-based EC2 instance within its Amazon Web Services (AWS) infrastructure. Upon instantiation, the student's submission, the instructor's test scripts, and relevant assignment metadata are transferred into the virtual machine (VM), where a `run_autograder` bash script executes to produce the results. The graded results are saved to a `results.json` file and passed back to the Gradescope client before the VM is destroyed [15].

Due to the constraints of Gradescope's design, all autograders must run within its managed AWS VMs. Therefore, *GreenGrader* was designed to offload as much computation as possible onto the Junkyard Computing cluster while maintaining compatibility with Gradescope's execution model. Within this model, the *GreenGrader* `run_autograder` script compresses the student's submission into a ZIP archive and issues a `curl` command to a job server running on the Junkyard Computing cluster to schedule the grading task. Upon successful job creation, the job server returns a job ID, which the VM polls at five-second intervals for the job's status. When the job successfully finishes, the student's results are sent back to the VM and written to the `results.json` file. If the job server returns an error, or does not return a response after 300 seconds, the script terminates and assigns a score of zero for the student.

4.1.2 Job Server. This is a server that accepts requests from the outside world to create tasks (jobs) in the phone cluster. We call it the Job server. It runs on one of the worker nodes and is implemented in Go. Its functions are:

- Accept a student submission for an assignment from GradeScope.
- Configure the job that runs the autograder. This involves pulling the container image containing the environment and dependencies for the assignment and the autograder.
- Schedule a job that grades the code.
- Monitor the job.
- Return the results to GradeScope.

The highlight of the Job server is its ability to receive requests concurrently, schedule tasks, and monitor them asynchronously. This is done by ensuring that the job monitoring mechanism does not block the server from receiving new requests. This is an improvement of our initial implementation that would wait for the current request to finish and only then accept the next request.

The concurrency is implemented by by exposing two HTTP endpoints

- `/submit` - Receives a `POST` with the students' submission and proceeds to schedule a job to run the autograder for the submission. It creates a job, launches an asynchronous subroutine and returns a `job_id` to GradeScope. The subroutine monitors the job in the background while the Job server processes other requests. A major aspect of monitoring is the computation of latency.
- `/status/<job_id>` - GradeScope polls this endpoint periodically to check the status of its jobs until there is a timeout or the job is complete/failed. This endpoint exposes the monitoring mechanism.

Since multiple calls are made to the above subroutines, they are made thread-safe by safeguarding critical sections using mutexes.

4.2 Deployment and Results

The *GreenGrader* was deployed for a dummy course based on CSE160 that consists of OpenCL assignments. Since the GPU in the phone is not yet supported, we support OpenCL for CPU.

4.2.1 OpenCL Support. We use PoCL (Portable Computing Language) which is an open-source implementation of the OpenCL standard. To install POCL into our system, we utilized a pre-built image from the CSE 160 course, specifically a pre-built image containing the necessary OpenCL runtime and tools. We then created a Kubernetes Pod using a YAML script (opencl-cpu.yaml) that runs this image in a container named opencl-container. The pod is set to run indefinitely using the 'sleep infinity' command, allowing us to interact with the environment as needed. After applying the script with kubectl apply -f opencl-cpu.yaml, the pod is deployed and ready for testing OpenCL programs in a consistent and containerized setup.

After deploying the pod, we copied over assignment-specific files from CSE 160 Assignment 2, including the helper library, example solution files, datasets, and the provided Makefile. However, the original Makefile was not compatible with our arm64 architecture in the phone cluster. To address this, we modified the Makefile to

detect both the operating system and machine architecture using uname -o and uname -m. Specifically, we added a conditional block to check if the system is GNU/Linux and the architecture is aarch64. If so, the Makefile links against -lOpenCL directly. These changes allowed the code to compile and link correctly on our ARM system using POCL.

To make the assignment grading process more scalable and easier to maintain, we modified the Makefile by replacing the manually written execution commands with a loop in the run target that iterates over all datasets using ‘seq 0 9.‘ We added error handling by capturing the exit status of each run and logging whether the dataset passed or failed into a summary file (run_logs/summary.log). By capturing the exit status, we were able to run all the tests instead of causing the cluster to crash with a single error. If a run failed, the reason and corresponding log file were noted for easy debugging. Additionally, we introduced a grade target that parses the summary log to count the number of passed datasets and prints the final score along with a brief grading summary.

To further streamline the deployment process, we created a custom Docker container that extends from the original POCL container image. This new container includes all the necessary components: the modified Makefile with automated testing loops, the helper_lib directory containing the assignment’s helper files, and all the required datasets. Most importantly, I modified the output format to meet GradeScope’s parsing requirements by implementing a results.json file generation system. In addition to logging pass/fail status to text files, the Makefile writes the results of each test case to a results.json file that provides a score out of the total possible points. This JSON format allows GradeScope to automatically parse and grade the assignment result, providing a complete, self-contained environment that maintains compatibility with GradeScope’s grading infrastructure.

4.2.2 Results. The performance of the GreenGrader system is evaluated using two metrics - latency and throughput. The highlight of the system is its ability to concurrently schedule jobs across the cluster. The latency of a batch of jobs is computed as the time difference between the end of the last job and the start of the first job in the batch.

$$t_{batch}(seconds) = t_{end-last-job} - t_{start-first-job}$$

The throughput of a batch of N jobs is computed as follows

$$\text{Throughput (jobs/minute)} = \frac{N \times 60}{t_{batch}}$$

The throughput for varying batch sizes (number of jobs) is shown in Figure 7. These results were obtained for a CSE160 OpenCL assignment. The ability of the cluster to run jobs concurrently is evidenced by the throughput increasing with the number of jobs. Moreover, the throughput of the batch is higher than the throughput of an equivalent number of jobs running sequentially (4.28 jobs/minute). We draw two main conclusions from these results

- The throughput increases with the number of worker nodes. However, this behavior is not significant when the number of jobs is low because workers are underutilized. This can be observed in the throughput difference between the two graphs in Figure 7.
- The throughput increases with the number of jobs and is expected to increase until the system is saturated. We were unable to find the saturation point due to GradeScope’s limitation of not supporting programmatic submission of assignments. In other words, we had to manually submit assignments for the dummy students in the course to test the system in a real-world setting.

Fig. 7. Performance results for GreenGrader: Throughput w.r.t the number of jobs and the number of worker nodes. The throughput of the batch is higher than the throughput of an equivalent number of jobs running sequentially (4.28 jobs/minute)

4.3 Limitations and Future Work

Although the *GreenGrader* performs reliably on the phone cluster, there are a few limitations.

- Gradescope launches an EC2 instance within its Amazon Web Services (AWS) infrastructure by default. As a result, the instance is almost idle and waiting for the GreenGrader to complete its tasks. This is unnecessary and inefficient. However, this behavior cannot be avoided and we therefore propose to explore alternatives like Github Classroom in the future.
- Configuring the GreenGrader for grading assignments requires knowledge of the phone cluster and its infrastructure. Ideally, the system should provide user-friendly configuration that does not require understanding of the underlying design.

4.4 Milestone Status

- Working Kubernetes Setup with multiple phones - Complete:
  - We added configs, kernel patches and flashing process for all 16 Pixel Fold phones, giving us a cluster with one master node and other worker nodes
  - Ephemeral-storage and memory requests tuned, eliminating pod-eviction
- Web-service exposure - Complete: The custom `nginx:alpine` image has been replaced by the Job Server; traffic now flows through a Kubernetes Service + NodePort and an Apache reverse-proxy on *Automaton*, giving a single public endpoint (`smartcycling.sysnet.ucsd.edu/gradescope`)

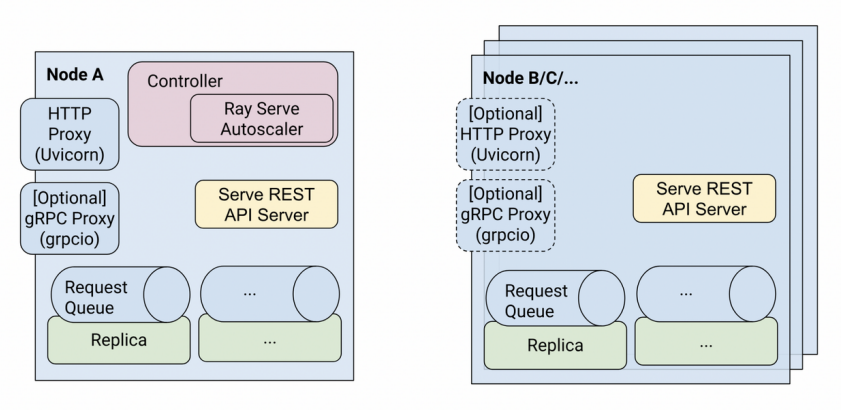

Fig. 8. An architecture diagram of a Ray cluster. Each node represents a node on the cluster while each replica represents a worker on that node. The controller automatically scales up and down replicas.

- System Architecture Documentation - Complete: Final diagrams have been added to the report and our work has been documented on GitHub and Notion
- GreenGrader MVP - Complete:
  - Integrated with Gradescope for an asynchronous Job Server that accepts a submission, schedules a Kubernetes Job, and the client polls until completion.
  - Achieved concurrency using the Goroutine so that the cluster grades the test assignments in parallel.
  - Created a metrics pipeline using `metrics.go` to writes a cumulative `latency_state.json` inside the container with our total latency results for the jobs.
- Dropped / modified milestones:
  - Using *Automaton* as the Kubernetes master was abandoned (DNS incompatibility), but this had no impact on functionality since a phone as a master has proven sufficient.

The *GreenGrader* has achieved all the milestones that were proposed at the beginning of the project!

5 FishSense

This section discusses the design of the FishSense system in detail. We start by describing the design at a high-level and then explain each component in detail.

5.1 System Design

For FishSense, the setup differs slightly from the GreenGrader setup. This setup uses the same Kubernetes cluster on the same phones as GreenGrader, but the key difference is that FishSense focuses on the setup and usage of Ray. According to Ray themselves, Ray is a tool that "precisely orchestrates infrastructure for any distributed workload on any accelerator at any scale." In our case, this means that Ray can be used to distribute high performance computing (HPC) workloads across the multiple phones in the cluster efficiently. On top of Ray's scalability, it is a) easy to install onto a Kubernetes cluster with Helm (Kubernetes' package manager), b) easy to write code for, and c) introduces built-in fault tolerance (if a worker dies, as long as the head isn't dead it will spin up a new worker).

In figure 8, each phone is represented by a node. FishSense is programmed to create multiple workers, each independently processing images in parallel. Since FishSense has a hefty memory usage requirement and each

phone only has 12 GB of RAM, each node/phone realistically only has 1 replica. Input files are brought into the system from a remote file system into the cluster via gRPC through the gRPC Proxy in figure 8.

5.2 Deployment

To successfully deploy the FishSense code from the FishSense Lite repo was the goal of the work. We were able to do this by installing onto the cluster via Helm a Raycluster with parameters that pointed to the Docker image produced by the FishSense Lite CI/CD pipeline.

```
helm install raycluster kuberay/ray-cluster \
  --version 1.1.0 \
  --set image.repository=ghcr.io/ucsd-e4e/fishsense \
  --set image.tag=ray-cpu \
  --set head.resources.limits.cpu=4 \
  --set head.resources.limits.memory=10G \
  --set head.resources.requests.cpu=1 \
  --set head.resources.requests.memory=10G \
  --set worker.resources.limits.cpu=4 \
  --set worker.resources.limits.memory=10G \
  --set worker.resources.requests.cpu=1 \
  --set worker.resources.requests.memory=10G \
  --set head.ray_start_params.num-cpus=0 \
  --set worker.minReplicas=2 \
  --set worker.replicas=2
```

Unfortunately, trying to install it directly without any further parameters caused the cluster to crash occasionally. After a lot of investigation, many of the problems were a result of FishSense Lite being very memory-intensive and that the default configuration would try to schedule multiple workers on the same node. This would crash the head of the Ray cluster and therefore fail the job (it cannot autoscale and replace failed workers if the head itself is dead). To fix this, we had to request CPU and RAM in a way such that each worker would be scheduled on a different node. This allowed the program to run successfully without crashing (too much).

A link to the deployment demo can be found at this YouTube link: youtu.be/OkqQY8ni7jQ. Images from the demo can be seen above. In figure 9 we can see the remote filesystem on the left and the terminal logged into the cluster on the right. In the terminal, there are 4 phones on the cluster and, each on a different node, 1 head and 2 workers for the FishSense Raycluster. In figure 10 we can see on the left side the remote file system with the appropriate outputs and, on the right that the process is fully complete.

5.3 Limitations and Future Work

There were two limitations to this project. The first is the lack of RAM on the phones. For normal computational workloads, 12GB of RAM should be enough to finish the job. For HPC workloads, however, there is a heavier emphasis on the usage of RAM, meaning that it is possible to run out of RAM very quickly. Future work would investigate how to mitigate this, since we can't exactly just increase the amount of RAM that a phone could have. We have looked into using swap to supplement the RAM, but for reasons unknown, despite having swap enabled in the configuration of the cluster, the swap space wasn't being used at all. Another limitation was the lack of GPU access for the phones. Since the FishSense Lite code was not tested to run on a CPU, there were many bugs in the code. This, however, is something that is a work in progress, and therefore may be accessible in the near future.

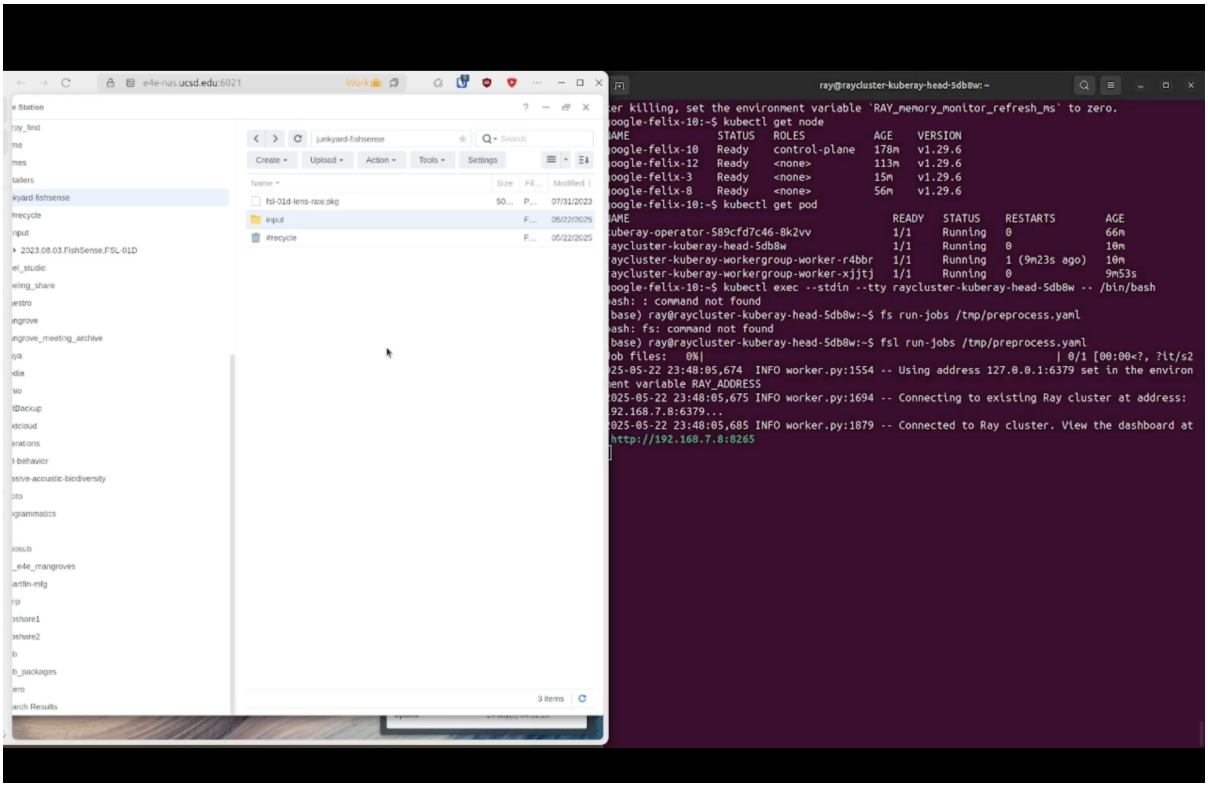

Fig. 9. Beginning of FishSense demo running

5.4 Milestone Updates

For FishSense, we were able to deliver on all of the milestones that we promised to deliver except for one, which was rendered obsolete. The goals that became unneeded were the goals that were related to setting up a web interface between FishSense in the cluster and outside of the cluster itself. This was because of the assumption that FishSense would receive input images from some form of web interface. This, however, was untrue since FishSense was able to retrieve the input files from an external remote file system. Because of this, we decided to stop pursuing goals related to setting up a web interface.

The first milestone directly related to FishSense was the design and documentation of the system architecture. In this milestone, there were three subtasks. Of these three subtasks, we were able to complete two: 1) Diagram out how Helm and Ray.io's CRD interact with the cluster and 2) Research and document how Kubernetes, Helm, and Ray.io interact with each other. We were able to accomplish both of these within the first half of the quarter. The second milestone that we worked on was the deployment of FishSense Lite to the cluster, a milestone that encompassed the setup and documentation of Ray (as well as the Hello World programs) on the cluster as well. We were able to successfully complete this within the last few weeks of the quarter.

6 Acknowledgments

We would like to express our sincere gratitude to the many individuals and organizations who supported this project - Prof. Kastner for giving us the opportunity to be part of the Junkyard project, Jennifer Switzer and

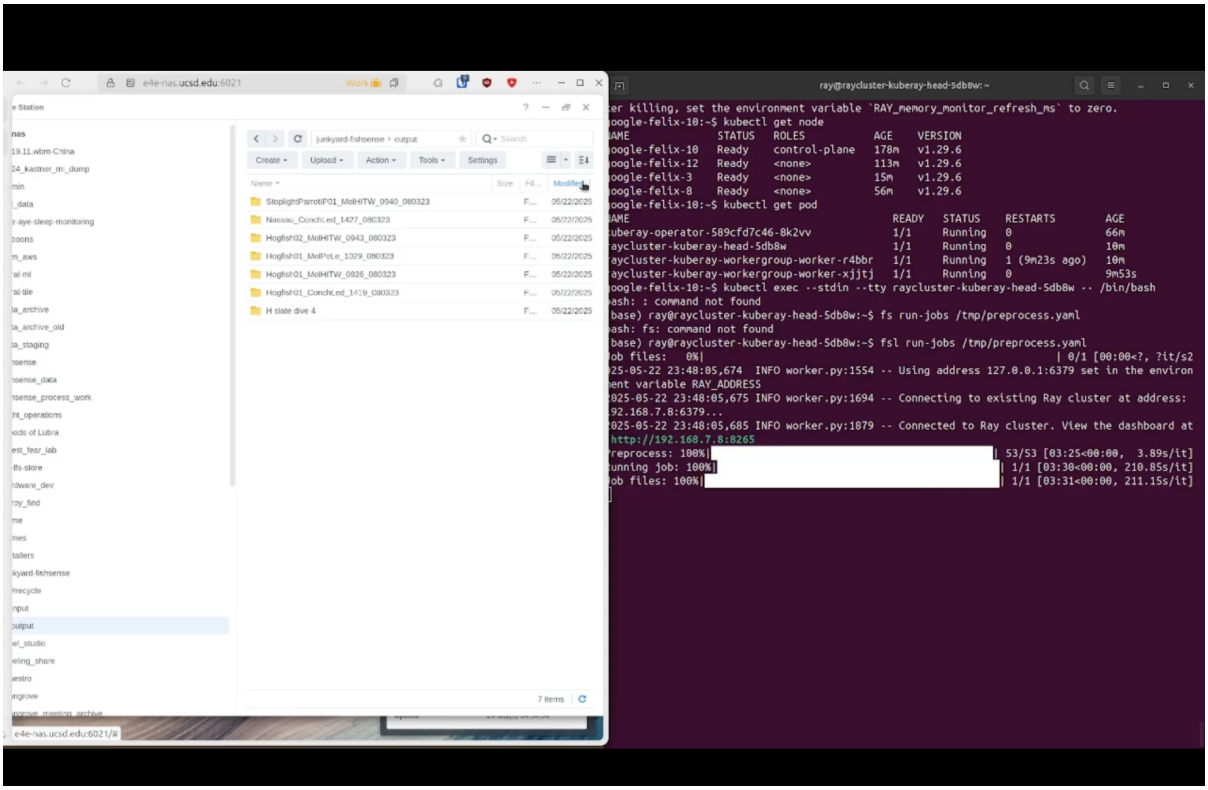

Fig. 10. End of FishSense demo

Christopher Crutchfield for their mentorship and willingness to engage in discussion that helped us improve our work, and finally, Google for providing many phones to enable testing and deployment of our system.

7 Conclusion

E-waste and lack of computing resources are both rapidly growing concerns. The Junkyard Computing project overcomes this challenge by offering a sustainable, low cost, phone-based data-centre, which can concurrently run autograding and ML workloads. Our setup consisted of Google Pixel Fold phones running PostmarketOS and Kubernetes with a container-based workflow (Docker + containerd). The *GreenGrader* uses a Go-based server that is capable of servicing and creating jobs concurrently. The server monitors running jobs by polling them. For FishSense, we used a Ray deployment for distributed computer-vision workloads. Overall, our results prove the feasibility of using smartphones as resources for general purpose computing.

The study also surfaced limitations: 12 GB of RAM per phone constrains large high-performace workloads; lack of open-source GPU drivers forces CPU-only usage; Gradescope's mandatory EC2 runner leaves a redundant VM idle during off-loaded grading. Addressing these issues and adding swap-aware schedulers, enabling GPU kernels, and integrating GitHub Classroom constitute the immediate future work. Longer-term directions include persistent metric storage, power-aware scheduling, and broader deployments in resource-constrained regions.

The Junkyard Project demonstrates that a cluster built entirely from discarded smartphones can perform real, end-to-end workloads that are normally completed on institutional data-centres.

8 Bibliography

[1] M. Y. Arslan, I. Singh, S. Singh, H. V. Madhyastha, K. Sundaresan, and S. V. Krishnamurthy. 2015. CWC: A Distributed Computing Infrastructure Using Smartphones. IEEE Transactions on Mobile Computing 14, 8 (Aug. 2015), 1587–1600. https://doi.org/10.1109/TMC.2014.2362753

[2] Giuseppe Massari, Martino Zanella, and William Fornaciari. 2016. Towards Distributed Mobile Computing. In 2016 Mobile System Technologies Workshop (MST). IEEE, Milan, Italy, 29–35. https://doi.org/10.1109/MST.2016.13

[3] Felix Büsching, Sebastian Schildt, and Lars Wolf. 2012. DroidCluster: Towards Smartphone Cluster Computing – The Streets are Paved with Potential Computer Clusters. In 2012 32nd International Conference on Distributed Computing Systems Workshops. IEEE, Macau, China, 114–117. https://doi.org/10.1109/ICDCSW.2012.59

[4] Pedro Miguel Costa Sanches. 2017. Distributed Computing in a Cloud of Mobile Phones. Ph.D. Dissertation. Universidade NOVA de Lisboa, Portugal.

[5] Jason Switzer, Gabriel Marcano, Ryan Kastner, and Pat Pannuto. 2023. Junkyard computing: Repurposing discarded smartphones to minimize carbon. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 2. ACM, Vancouver, BC, Canada, 400–412.

[6] Shane Ward and Mark Gittens. 2018. Building useful smart campus applications using a retired cell phone repurposing model. In 2018 Third International Conference on Electrical and Biomedical Engineering, Clean Energy and Green Computing (EBECEGC). IEEE, Beirut, Lebanon, 43–48. https://doi.org/10.1109/EBECEGC.2018.8357131

[7] Ulrich Norbisrath, Patrick Mukala Ngoy, Rui Pedro Machado, Huber Flores, and Madhusanka Liyanage. 2025. Empowering Sustainability: Upcycling Smartphones as the Future of IoT and Edge Computing in Emerging Economies. In Proceedings of 23rd International Conference on Informatics in Economy (IE 2024) (Smart Innovation, Systems and Technologies, Vol. 426), Cristian Ciurea, Paul Pocatilu, and Florin Gheorghe Filip (Eds.). Springer, Singapore, 145–170. https://doi.org/10.1007/978-981-96-0161-5$_1$8

[8] Jason Switzer, Emily Siu, Srivatsan Ramesh, Richard Hu, Eamon Zadorian, and Ryan Kastner. 2022. Renée: New Life for Old Phones. IEEE Embedded Systems Letters 14, 3 (Sept. 2022), 135–138. https://doi.org/10.1109/LES.2022.3147409

[9] Josh Suckling and Jacquetta Lee. 2015. Redefining scope: the true environmental impact of smartphones? The International Journal of Life Cycle Assessment 20 (2015), 1181–1196. https://doi.org/10.1007/s11367-015-0909-4

[10] Edgar Nett. 2004. An architecture to support cooperating mobile embedded systems. In ACM International Conference on Computing Frontiers (CF'04). ACM, 40–50.

[11] Danyi Liu, Adnan Al-Anbuky, and Ching Lin. 2007. Mobile embedded database for remote process management system. In 2007 Australasian Telecommunication Networks and Applications Conference. IEEE, Christchurch, New Zealand, 515–520. https://doi.org/10.1109/ATNAC.2007.4665231

[12] David von Leon, Lorenzo Miori, Julian Sanin, Nabil El Ioini, Sven Helmer, and Claus Pahl. 2018. A Lightweight Container Middleware for Edge Cloud Architectures. In Fog and Edge Computing: Principles and Paradigms. Wiley, 145–170. https://doi.org/10.1002/9781119525080.ch7

[13] Dingji Du, Qingyuan Liu, Xiangrui Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ACM. https://doi.org/10.1145/3503222.3507732

[14] Panos Patros, Josef Spillner, Alessandro Vittorio Papadopoulos, Blesson Varghese, Omer Rana, and Schahram Dustdar. 2021. Toward Sustainable Serverless Computing. IEEE Internet Computing 25, 6 (Nov.-Dec. 2021), 42–50. https://doi.org/10.1109/MIC.2021.3093105

[15] Gradescope. 2022. Gradescope Autograder Documentation: Technical Details (August 2022).