Owner-centric Smartphone Lifetime Extension

Justin Hudis
jch2248

Synopsis

Overview

This project is a follow-up to my midterm paper on current strategies for smartphone lifetime extension (SLE). In that paper, I discussed the growing problem of frequent phone replacement which creates an excess of carbon emissions. I also ran an experiment to see whether a smartphone owner could profit from keeping their old phone charged while running a bandwidth-sharing app. I concluded that the problem of SLE can be alleviated with investments in repair, refurbishment, durability, and especially upcycling, while support for recycling, modularity, and phones-as-IoT is far less promising. I also found that the payout from a bandwidth sharing app is indeed orders of magnitude greater than the cost of running one.

These results inspired me to take an owner-centric approach to smartphone upcycling. Essentially all current research on upcycling assumes that, eventually, smartphone providers or manufacturers (whom I will henceforth refer to together as simply "providers") will design server farms which will house towers of interconnected smartphones that work together to provide services from AI training to web hosting. This is a major assumption that has not been addressed head-on in literature. It is not clear that providers have enough of an incentive to design such a server farm, even if its practicality is proven in literature. It may turn out to be less profitable than investments in marketing and new designs. It may be adopted in the distant future, but every year that passes more phones will become unusable. We ought to try building an immediate solution that doesn't rely on providers aligning their interests with ours.

Functionality and Design

To that end, I have developed a system which allows provisioners to execute their code on old Android phones which are running an app I wrote. Figure 1 shows a model of this system. The provisioners make requests to the hub, the hub distributes those requests to the workers (phones), the computation happens on the phones and the result is sent back up to the provisioners.

In my experiments, unless otherwise stated, I used one provisioner and four workers. My personal laptop served as a provisioner and four Motorola Moto E4 Plus XT1775 phones served as workers. I used a 4 GB RAM / 2 vCPU / 80 GB SSD DigitalOcean Droplet as my hub.

My system assumes that the average smartphone owner is "lazy." That is, they don't want to buy new peripherals like smart chargers, they don't want to figure out how to replace their batteries, they don't want to do technical setup, etc. This makes for a less efficient system, but widespread adoption is the priority. Once that is achieved, there is potentially room to educate active users through the app about how to use it more effectively and perhaps provide monetary incentives.

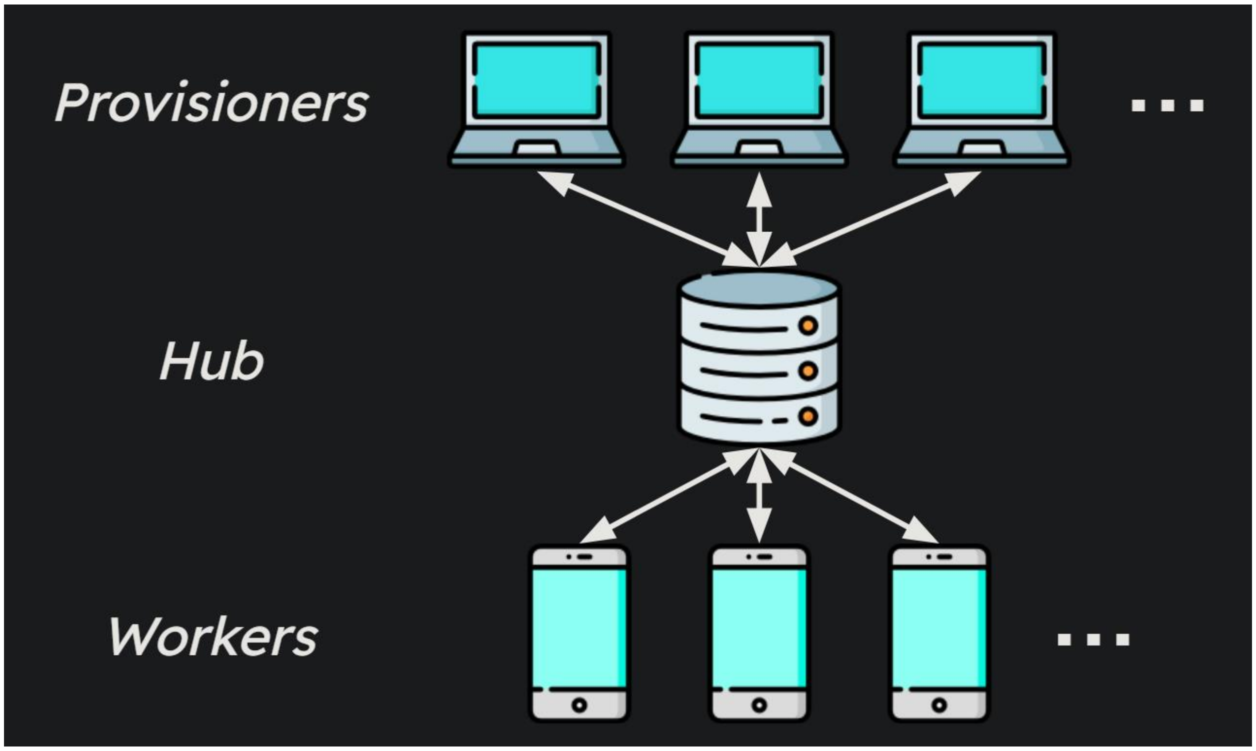


Figure 1: Model of the system I designed.

Figure 2 shows the system in action. I had four phones fulfilling requests while a fifth phone monitored their status. A more realistic scenario would be someone running jobs on their old phone while monitoring it on their new phone. That said, an owner may want to monitor more than one phone if they are a parent monitoring all the old phones in their household.

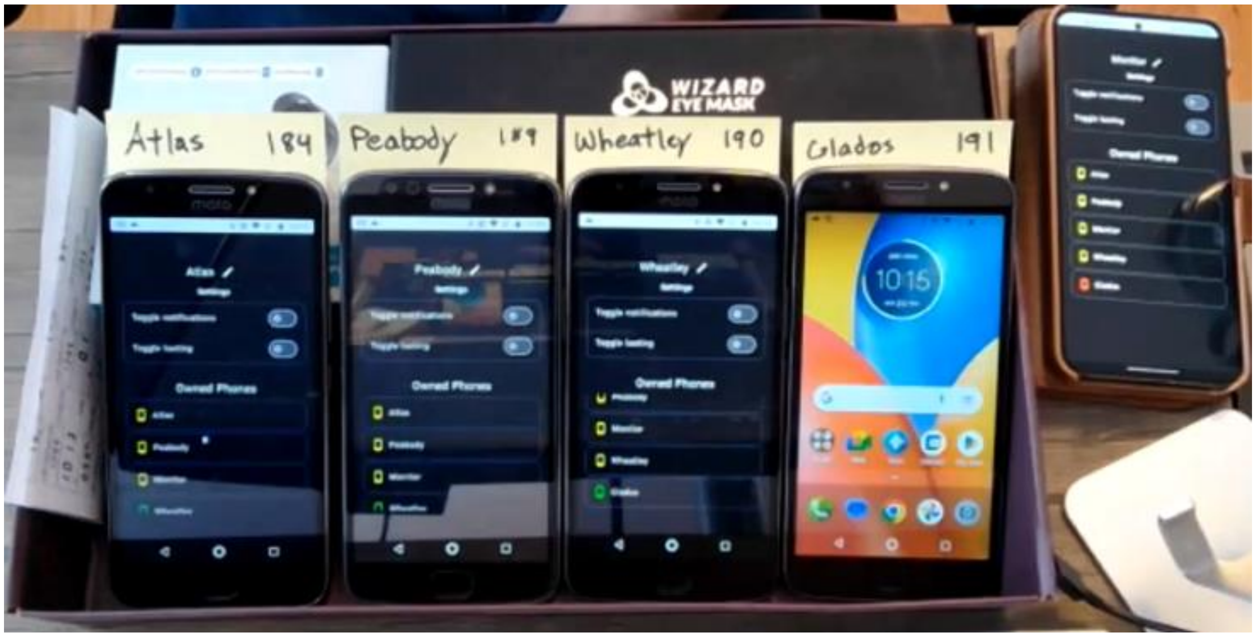


Figure 2: A screenshot from my demo. At this moment, the worker phones were set to idle and one of them was manually closed to show that the monitor would detect an issue with the device.

Provisioners upload and invoke functions using REST calls which trigger writes to WebSocket connections between the hub and phones. The phones register themselves in the database and keep their displays up to date by making REST calls to the hub, which stores the status of all phones. In the future, a pub/sub model may be more appropriate. With additional time, I also would have added notifications so that an owner could be informed when their old phone loses connection or experiences a crash.

Alternatives

There do already exist apps on the Play Store that an owner can download on their old phone to give it a second life.

There are bandwidth sharing apps, which I studied in my midterm paper. These use the same model as my app, except that the only resource these apps consume is network bandwidth; the RAM, CPU, and storage of the phone is still going to waste. This is partly by design, since the primary use case is someone who wants to make a little extra money by running something in the background on their current phone, not an old one. Thus, it would be a problem for the app to use up considerable resources and slow down the phone. It is also more convenient for the app developers and maintainers (whom I will henceforth refer to together as simply "developers") because it reduces the scope of their app significantly. There are fewer security, compatibility, design, and marketing concerns to deal with when the only functionality is traffic routing.

There are cryptocurrency mining apps, which again have a similar model to my app but are more specialized. While I have not tested the profitability of using such an app, I suspect that it will be far less profitable than bandwidth sharing because it has to compete against specialized hardware that is already mining much faster. In addition, cryptocurrency is limited in scale and largely seen as detrimental to the environment, which not only poses a problem in itself but also may deter owners from adopting the application.

Lastly, there is the large category of phone-as-IoT applications. There are apps that turn your phone into a security camera, a second monitor, a PC resource monitor, a health monitor, etc. These solutions may work for some individuals, but they aren't scalable. Not everyone wants a miniature second monitor or has the tech savvy to set it up. Additionally, most of these solutions also do not fully utilize an old phone's resources.

Innovations

The novelty of my app is that it is general-purpose. Any provisioner can run any Java 8 code they want as often as they want, with the only limitations being the number of phones available and the desired latency. General-purpose compute resources are highly valuable and have the potential to consume a phone's resources far more thoroughly than previous solutions.

My app differs from general-purpose solutions in literature because it does not rely on the phones all being in the same physical space. While this does increase latency, it also means that cooling is unnecessary and owners are responsible for charging and maintenance, eliminating all the additional monetary and environmental costs that are often noted in academia.

Value

There are four stakeholders in this context: the owners, the developers, the provisioners, and the environment. Notice that the providers have been taken out of the equation. The owners and developers want to make money, the provisioners want to spend less money than they usually would, and the environment, so to speak, wants carbon emissions from manufacturing to not go to waste. Later in this paper, I will show that my system allows provisioners to pay substantially less for cloud services equivalent to what is currently on the market while paying the developers and owners enough and being better for the environment than current services.

Availability

My app is compatible with Android phones running Android 5.0 or higher. This is the lowest Android Studio let me choose and it claims to comprise 99.6% of devices, which I see as sufficient. Surely some people may still have older phones than that, but it is rare that anyone would keep a phone that long and that it would still be functional and powerful enough to be worth using. iOS support would be ideal but would require building another app from the ground up. Perhaps it would have been wise to pursue cross-platform development, but since this is proof of concept, I figured it would be safer to use a dedicated development platform.

Ultimately, I would need to get my app on the Play Store for it to be adopted. A user would currently have to clone my GitHub repository and build the app themselves in Android Studio, which obviously no one would do. Getting my app on the Play Store was unfortunately out of scope for this project, however, since the approval process takes time, and my app currently has too many security concerns to be approved anyway.

Security

Anyone with any knowledge of cybersecurity knows that allowing someone to run arbitrary code on your machine is dangerous. The code is run using BeanShell on Android, a combination which does not appear to have been investigated for vulnerabilities but certainly has some. Depending on BeanShell's capabilities, a malicious user could potentially read files, open someone's camera, listen to their microphone, or worse.

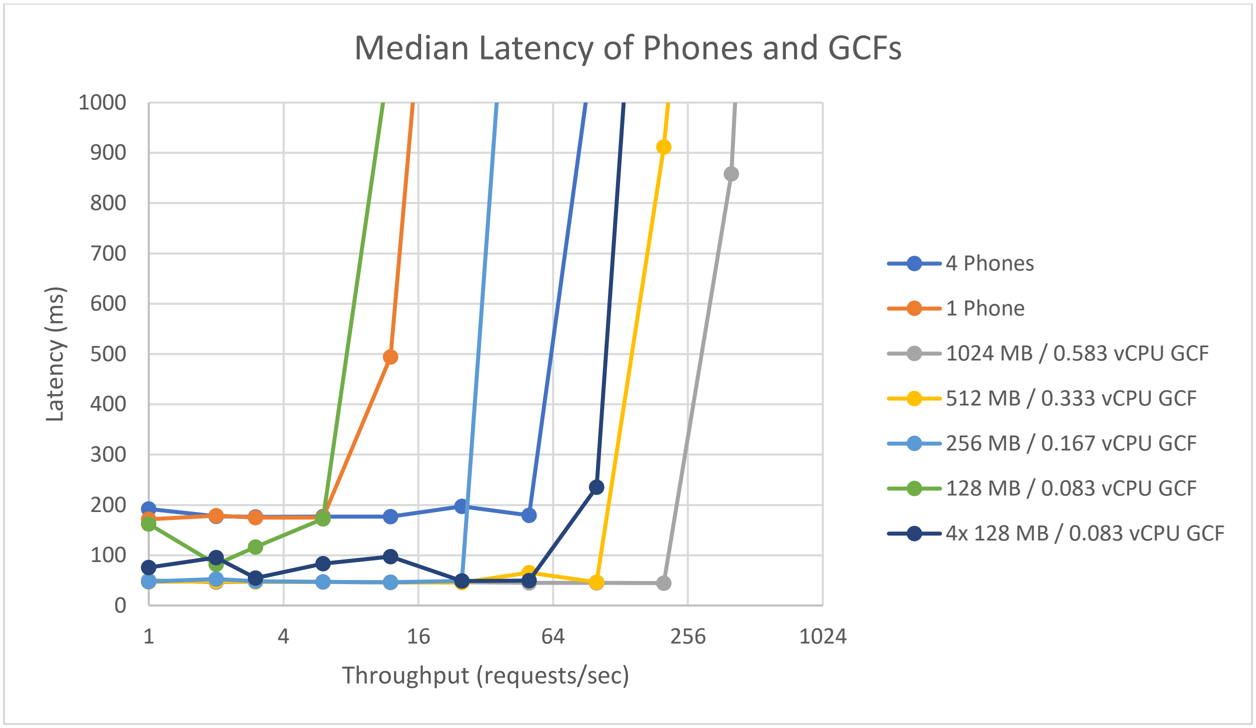
One way to deal with this would be to manually approve any code that a provisioner asks to run. While this model has been shown to work with bandwidth-sharing apps, as they approve whom they share bandwidth to, it may prove to be draining on human resources and some malicious programs could slip through the cracks. A better but more difficult solution would be to block certain commands (such as reading/writing to data outside of a space provided to the app) or run all code in a virtual machine that has been thoroughly tested for vulnerabilities. This was also out of scope for my project but would be absolutely necessary in practice.

Research Questions

Q1: Google Cloud Function (GCF) Latency Comparison

What Google Cloud Function (GCF) configuration has similar latency to my phone cluster?

To answer this question, I ran the same Java code (prime testing) on my phones and GCFs of various configurations, increasing the throughput of requests over time. Each throughput was tested for 5 seconds, then doubled, until the system could no longer keep up. The median and tail (90th percentile) latency were measured for each throughput. The results are shown in Figure 3.



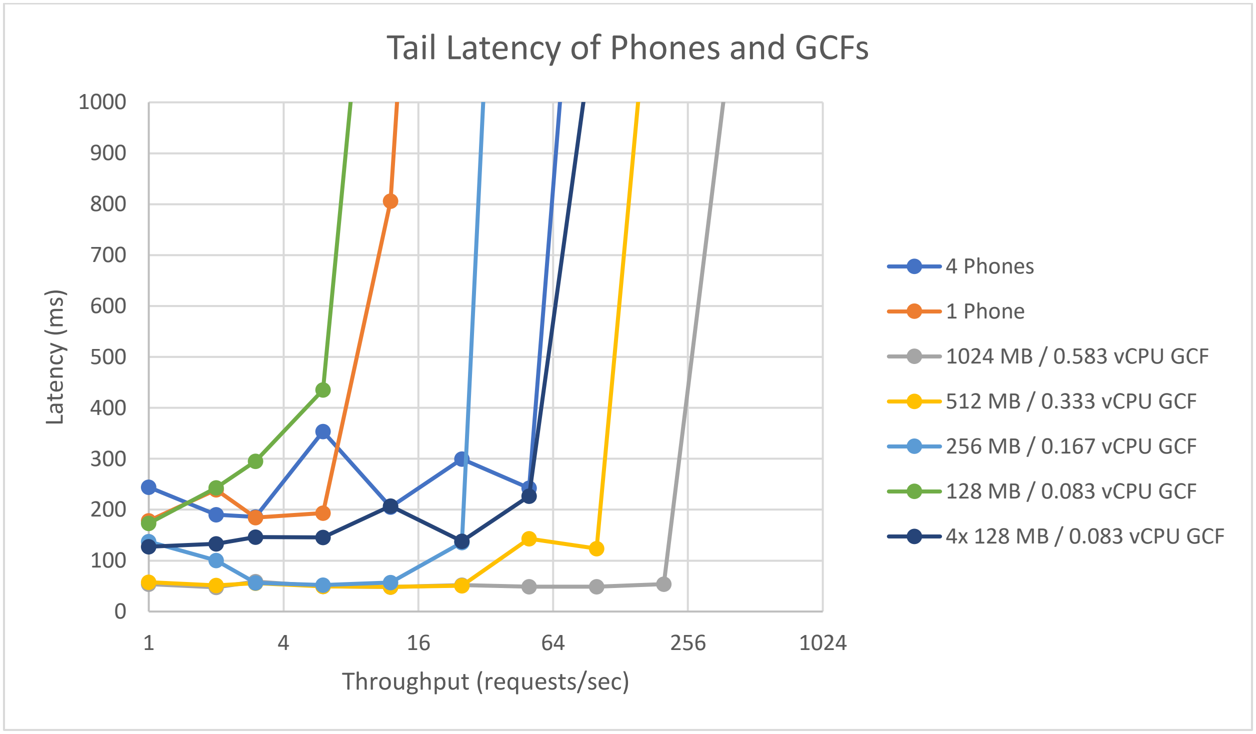


Figure 3: Median and tail latencies of phones and GCFs.

In terms of maximum throughput, it appears that my four-phone cluster is comparable to a 128 MB / 0.083 vCPU GCF with 4 instances. Interestingly, it underperforms the 1024 MB / 0.583 vCPU GCF despite having nearly identical hardware specifications, and in almost every case the phones have poor low-throughput latency. Presumably this is because Google's hardware and network architecture is optimized for these tasks and because my Android app is surely not allowed to consume the total resources of the phone.

As an ablation study, I also ran this system with all but one phone set to idle. The maximum throughput indeed went down substantially, and interestingly the overall performance is comparable to a single 128 MB / 0.083 vCPU GCF instance. This is a good result because it shows that the bottleneck lies within the number of phones, not the bandwidth of the hub, and that additional phones produce exponential benefits.

Going forward, "GCF" will refer to the 4x configuration.

Q2: Cost Comparison

Is my phone cluster significantly cheaper to run than GCF?

This question is important because if its answer is no, then this concept is not practical. Environmental benefit is not enough of an incentive for provisioners to switch to another service; it needs to be considerably cheaper for the switch to be worth it. Thankfully, it is. If each system were run continuously at near maximum capacity, i.e. 50 requests per second, after a month the phones would cost $0.34 in electricity while Google would charge $81.81 for their service. See Cost Calculations for details.

Note that I am leaving networking out of the phone cost calculations. This is because it is extremely rare for an owner to pay for Wi-Fi by usage. Rather, a household almost always pays a fixed rate per month based on the bandwidth they're paying for. Thus, it does not cost an owner extra on their internet bill to run my app on their phones. If an owner's Wi-Fi has very low bandwidth, they may be using that up by running my app and notice performance drops on their other devices, but even with four phones running I have never seen an issue myself. Weak, old phones usually don't have the ability to consume a considerable amount of bandwidth.

Moreover, I am leaving the cost of the hub out of my calculations. This is because, while I happen to pay $24 a month for my DigitalOcean Droplet, any number of machine types, both virtual and physical, could be used instead. Also, it isn't as if increasing the number of invocations directly increases the cost of running the hub, but rather the size of the hub is a bottleneck on the number of phones (or providers/requests) it can handle. As we saw in the previous section, however, that bottleneck has not been reached. It would take much further study to determine a relationship between the endpoint count and the cost of running the hub, which is out of scope for this project. Regardless, even if one were to be very strict and add $24 to the phones' cost per month, it would still outperform GCF substantially.

Q3: Profit

Is there a rough business model that would allow provisioners to pay less than current market prices while developers and owners profit?

Based on the numbers in the previous section, the answer is clearly yes. A more business-minded organization would need to determine a realistic model, but I can put together a rough one as follows. Charge the provisioners $40 a month, pay $30 to the developers, and pay the remaining $10 to the owners. The provisioners get to pay half what they would with standard cloud services, so they are happy. The developers can pay for the hub and have money left over (and consider this is only the profit from four phones), so they are happy. The owners make a bit less than $10 a month, so they are happy.

My prior research showed that the average phone lasts around 1-2 years longer than the amount of time its owner uses it. Typically, the battery is the first to go, and while they are usually replaceable with professional help, the average owner won't bother. At that point, we can be satisfied that the owner extracted another 1-2 years of use out of their phone and hope they send their phone in for refurbishment or at least recycling. Regardless, the tagline "make $200 before you throw your phone away" should entice owners to install my app.

Q4: Carbon Footprint Comparison

Is the Computational Carbon Intensity (CCI) of my system lower than GCF's?

While it's great that this system seems to be economically viable, the purpose of this field is to help the environment. To that end, we must ensure that this system does in fact have a lower carbon footprint than GCF. After all, these phones and my networking and charging setup are not optimized for cloud computing and are thus far less energy efficient. Following in the footsteps of Schwitzer et al., I computed the CCI over time for my system and for GCF. CCI can be summarized as the carbon emissions per operation averaged over a system's entire lifetime, and it should be minimized. Figure 4 shows the results. See Carbon Calculations for details.

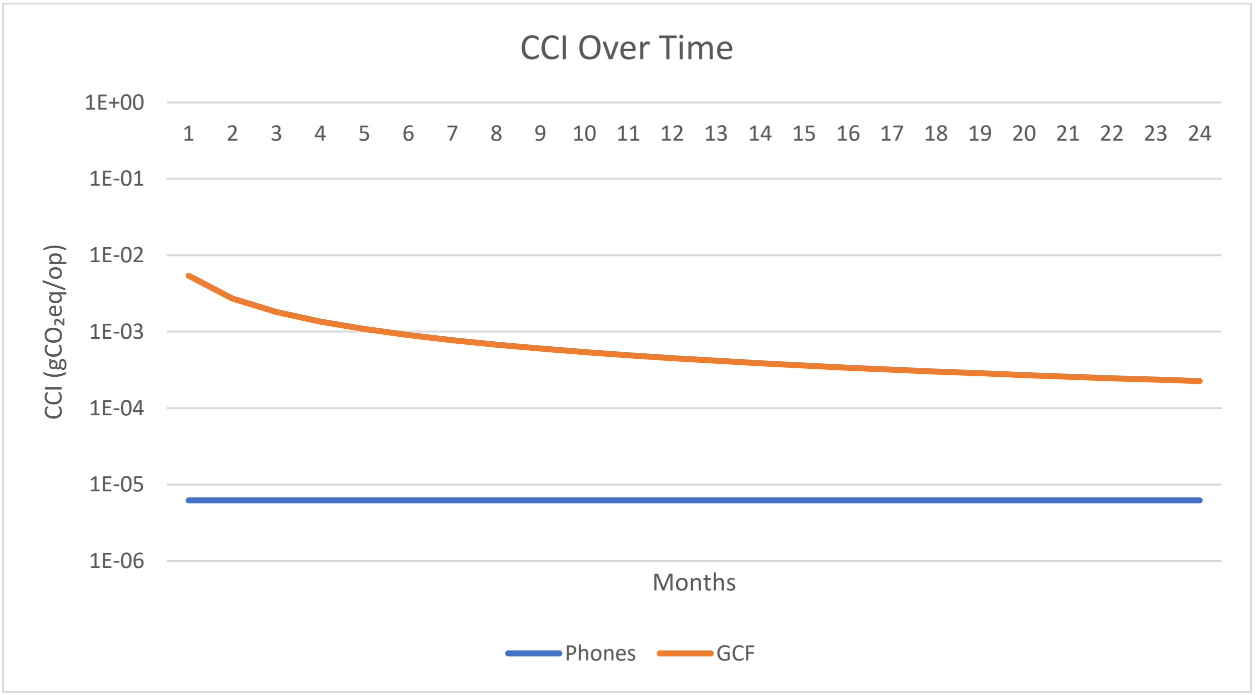


Figure 4: CCI over time for the phones and GCF.

Clearly, the phones outperform GCF in terms of CCI over time. The advantage of my system is that it requires no new hardware, which would cost significant carbon emissions to manufacture. In fact, the "Carbon From Manufacture" value for my phone system is zero, since as Schwitzer et al. point out, this hardware normally gets thrown out, so using it in any way is, in essence, carbon-free. I'm excluding the hub's carbon from these calculations for similar reasons to earlier; it can vary wildly and does not scale directly with traffic.

Deliverables

The code for my project is contained in two GitHub repositories. The first repository contains the code for the Android app. The second repository contains code for the hub and testing/provisioner code, as well as all other requested deliverables.

Self-Evaluation

Accomplishments

I was able to develop an Android app in Kotlin and an Express app in Typescript and network them together. I debugged my system and presented a demo of it. I empirically determined the efficacy of my system and concluded that it is practical.

Challenges

One of the biggest challenges I faced immediately upon starting this project was discovering that Android development had a complete overhaul in the last few years. I haven't developed for Android for a few years, and in that time the entire framework changed. XML is gone, Java was replaced with Kotlin, libraries work differently, basically everything was new. I had expected to only need to catch myself up on Android development, but I actually had to watch and read tutorials for each new thing I wanted to do.

Concurrency was another big challenge for me. Android and Typescript are both environments where blocking functions on the main thread are frowned upon, where LaunchedEffects and setTimeouts are used liberally. I've never received a formal education in concurrency and this style of coding was somewhat foreign to me.

Finding a library that could run arbitrary Java code on Android was a very long struggle, and I almost gave up. I finally found BeanShell and got it to work, but it was far from the first search result and even when I first found it I couldn't tell that it was what I wanted.

Trying to compute the carbon emissions of a proprietary system is nearly impossible. Most of my calculations are rough estimates based on other people's rough estimates, and even those took ages to find.

Learning Outcomes

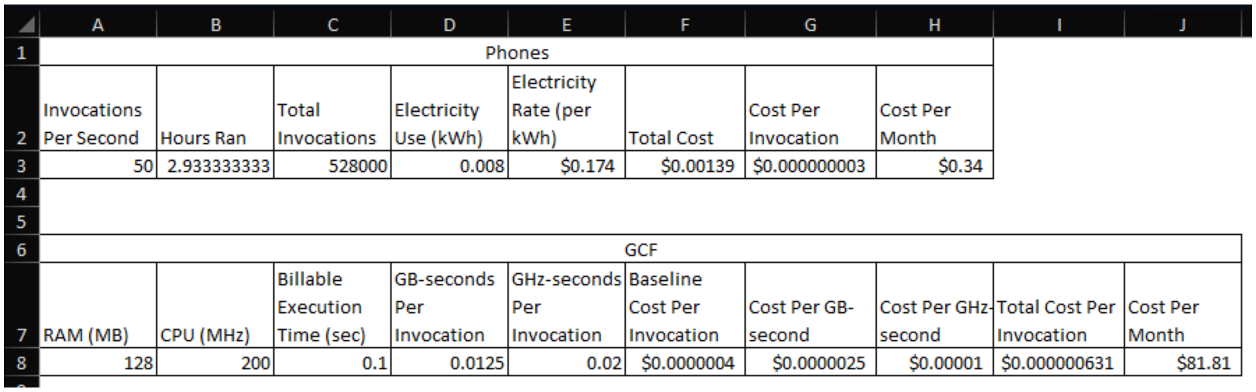
I learned the new Android framework and became more familiar with Express. I learned the order of magnitudes of monetary and environmental costs of computation. I concluded that the system I've been thinking about this entire semester is in fact viable.

Appendix

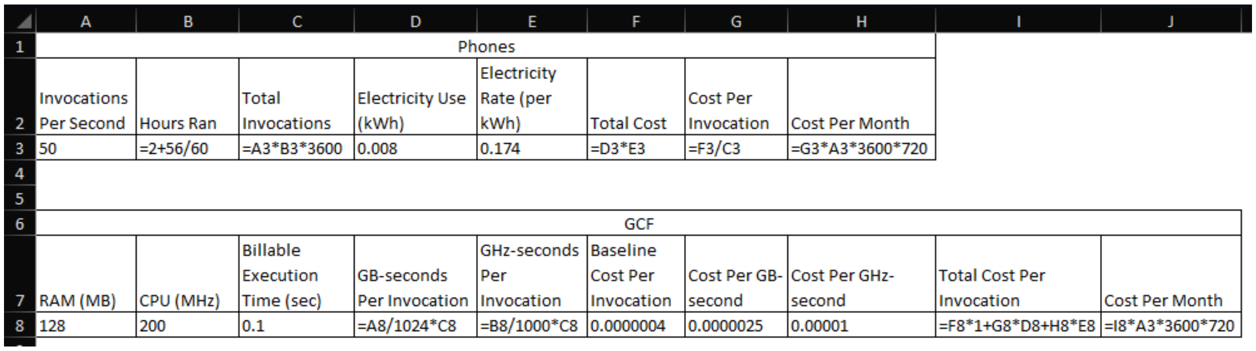
Cost Calculations

Values and formulas are either empirical/standard or come from the following sources:

- Average U.S. Energy Prices: E3
- GCF Pricing: D8-I8



	A	B	C	D	E	F	G	H	I	J
1	Phones									
2	Invocations Per Second	Hours Ran	Total Invocations	Electricity Use (kWh)	Electricity Rate (per kWh)	Total Cost	Cost Per Invocation	Cost Per Month		
3	50	2.933333333	528000	0.008	$0.174	$0.00139	$0.000000003	$0.34		
4										
5										
6	GCF									
7	RAM (MB)	CPU (MHz)	Billable Execution Time (sec)	GB-seconds Per Invocation	GHz-seconds Per Invocation	Baseline Cost Per Invocation	Cost Per GB-second	Cost Per GHz-second	Total Cost Per Invocation	Cost Per Month
8	128	200	0.1	0.0125	0.02	$0.0000004	$0.0000025	$0.00001	$0.000000631	$81.81



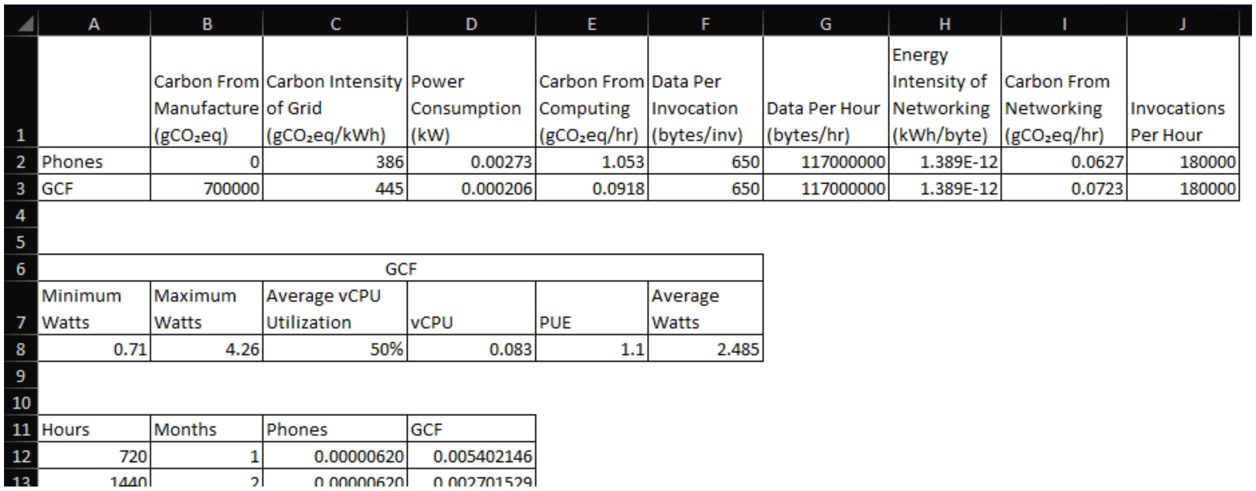
	A	B	C	D	E	F	G	H	I	J
1	Phones									
2	Invocations Per Second	Hours Ran	Total Invocations	Electricity Use (kWh)	Electricity Rate (per kWh)	Total Cost	Cost Per Invocation	Cost Per Month		
3	50	=2+56/60	=A3*B3*3600	0.008	0.174	=D3*E3	=F3/C3	=G3*A3*3600*720		
4										
5										
6	GCF									
7	RAM (MB)	CPU (MHz)	Billable Execution Time (sec)	GB-seconds Per Invocation	GHz-seconds Per Invocation	Baseline Cost Per Invocation	Cost Per GB-second	Cost Per GHz-second	Total Cost Per Invocation	Cost Per Month
8	128	200	0.1	=A8/1024*C8	=B8/1000*C8	0.0000004	0.0000025	0.00001	=F8*1+G8*D8+H8*E8	=I8*A3*3600*720

Note that I am using the average electricity rate in the U.S.

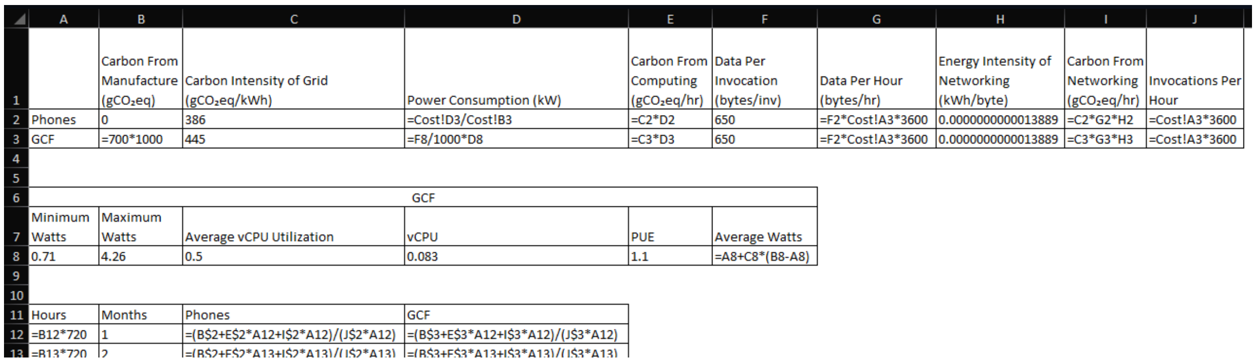
Carbon Calculations

Values and formulas are either empirical/standard or come from the following sources:

- AWS EC2 Carbon Footprint Dataset: B3
- Cutting carbon emissions on the US power grid: C2
- GCF Carbon Per Region: C3
- Junkyard Computing: E2-E3, H2-I3, C17-D35
- Cloud Carbon Footprint Methodology: D3, A8-C8, E8-F8



	A	B	C	D	E	F	G	H	I	J
1		Carbon From Manufacture (gCO$_2$eq)	Carbon Intensity of Grid (gCO$_2$eq/kWh)	Power Consumption (kW)	Carbon From Computing (gCO$_2$eq/hr)	Data Per Invocation (bytes/inv)	Data Per Hour (bytes/hr)	Energy Intensity of Networking (kWh/byte)	Carbon From Networking (gCO$_2$eq/hr)	Invocations Per Hour
2	Phones	0	386	0.00273	1.053	650	117000000	1.389E-12	0.0627	180000
3	GCF	700000	445	0.000206	0.0918	650	117000000	1.389E-12	0.0723	180000
4										
5										
6	GCF									
7	Minimum Watts	Maximum Watts	Average vCPU Utilization	vCPU	PUE	Average Watts				
8	0.71	4.26	50%	0.083	1.1	2.485				
9										
10										
11	Hours	Months	Phones	GCF						
12	720	1	0.00000620	0.005402146						
13	1440	2	0.00000620	0.002701529						



	A	B	C	D	E	F	G	H	I	J
1		Carbon From Manufacture (gCO$_2$eq)	Carbon Intensity of Grid (gCO$_2$eq/kWh)	Power Consumption (kW)	Carbon From Computing (gCO$_2$eq/hr)	Data Per Invocation (bytes/inv)	Data Per Hour (bytes/hr)	Energy Intensity of Networking (kWh/byte)	Carbon From Networking (gCO$_2$eq/hr)	Invocations Per Hour
2	Phones	0	386	=Cost!D3/Cost!B3	=C2*D2	650	=F2*Cost!A3*3600	0.0000000000013889	=C2*G2*H2	=Cost!A3*3600
3	GCF	=700*1000	445	=F8/1000*D8	=C3*D3	650	=F2*Cost!A3*3600	0.0000000000013889	=C3*G3*H3	=Cost!A3*3600
4										
5										
6	GCF									
7	Minimum Watts	Maximum Watts	Average vCPU Utilization	vCPU	PUE	Average Watts				
8	0.71	4.26	0.5	0.083	1.1	=A8+C8*(B8-A8)				
9										
10										
11	Hours	Months	Phones	GCF						
12	=B12*720	1	=(B$2+E$2*A12+I$2*A12)/(J$2*A12)	=(B$3+E$3*A12+I$3*A12)/(J$3*A12)						
13	=B13*720	2	=(B$2+E$2*A13+I$2*A13)/(J$2*A13)	=(B$3+E$3*A13+I$3*A13)/(J$3*A13)						

Since there are no manufacturing carbon estimates for GCF, I approximated by halving the emissions of the weakest EC2 instance (which is still about twice as powerful as a single GCF instance).

To expand on why I did not include the hub in my carbon analysis, consider that a single GCF instance has the same order of magnitude carbon emissions as my hub. A practical system would need many phones or many GCF instances; the first incurs no additional carbon, while the latter incurs significant carbon.