Moving Weird Gloop to GKE standard

Objectives

Make things faster

- Currently ~580ms to parse rs.wiki/Grimy_torstol
- 300µs round-trip time for redis
- ~50 minutes for rs.wiki Grand Exchange run
- GKE autopilot (current architecture) runs on E2 machines, which are fairly low-power and give about 30% less performance than some other machines offered on GCP.

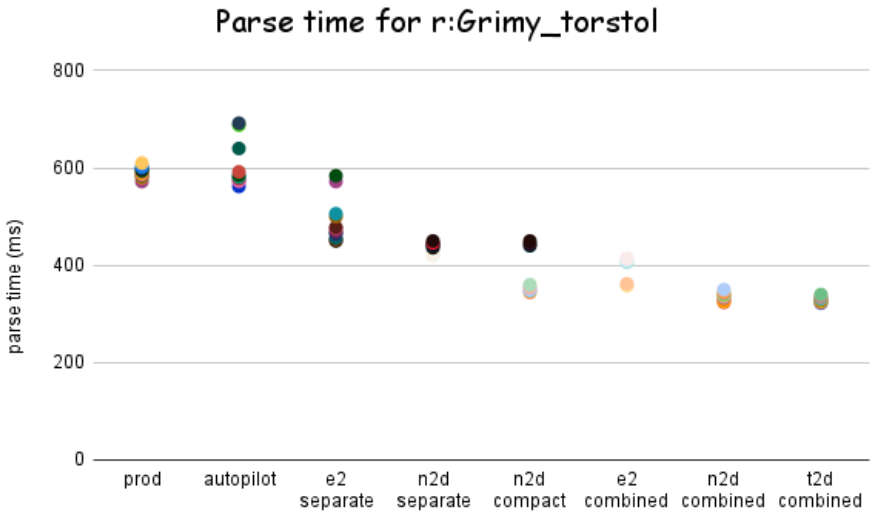
Reduce cost

- Currently $800/month (+ tax) just for GKE
- Even in the best case, GKE autopilot CPU/memory costs about twice as much as the same resources on GKE standard. While there is some additional Kubernetes overhead that we have to pay for on standard, this is vastly outweighed by the 50% cost reduction, better workload sizing (see my old post), and other upfront expenditures (mostly committed use discounts) we can make.

Improve stability (or at least don't make it worse)

- Generally in a pretty decent place now
- 40-60 seconds of downtime when redis-main-cache is being redeployed
- Previously had some memory wrangling issues with redis pods, seem to be mostly resolved
- mediawiki CPU spikes are much less common now due to reordering refreshLinks and htmlCacheUpdate jobs
- Would be good to at least allow some webserver autoscaling in case of badly-behaving scrapers (osrsbox, bingbot, etc)
- We are not resilient to zonal outages currently: while the database is high-availability and would failover (after some downtime) to a different zone, we are using a zonal (not regional) cluster and the stateful pods (redis, elasticsearch) probably wouldn't be operable until the zone returns.

Research questions



For parse time tests, we will be looking at the parse time of the "Grimy torstol" page on rs.wiki. This is totally arbitrary, but in general the relative parse time reductions we saw from other pages matched the reductions on Grimy torstol. Recall that the average parse time was about **580ms** on the existing architecture.

Does just moving from autopilot to standard improve parse time?

tl;dr: usually a bit, with some variation

We tried re-creating the production configuration on a new autopilot cluster, just to get a baseline ("autopilot" in the graph). The median parse time was about **570ms**, within about 1.5% of the production cluster, although from the graph you can tell there's a lot of variation, sometimes with the mean of 50 runs coming in as high as 700ms.

Then we tried to re-create the autopilot configuration as faithfully as possible in GKE standard, using the same low-power E2 machines that autopilot uses, and spreading the workload across a bunch of different nodes, so all of the relevant pods were on distinct nodes ("e2 separate" on the graph). Again, we saw significant variation in median parse time across different runs of 50, but the median of these runs was about **480ms**, about a 15-20% improvement over the autopilot test. The worst-case for the standard configuration was about the same as the average-case for autopilot.

The average parse time across different runs tended to not vary much once the node-pool was set up – rather, it seemed that the variability was mostly due to randomness in how the initial nodes were configured when they started running. I hypothesize that this is primarily due to some of the distinct nodes being created on the same bare-metal machine, reducing communication latency. So for the next test…

Does placing all of the pods on a single machine improve parse time?

tl;dr: YES, by around 20%, with some variation

mediawiki pods need to communicate frequently with redis pods to build pages, often 500 times per page parse. While it would be cool to batch more of this, MediaWiki is not really designed for that type of performance improvement right now.

In the current architecture, the time for a round-trip to redis is about 0.3 milliseconds. This doesn't sound like much, but because it happens many times per parse, it's a significant portion of the total page parsing time.

We know from prior research that in GCP, the round-trip time from one machine to another in the same local network is about 0.3ms. Can we do better if it's on the same physical machine and doesn't have to travel over the network?

We set up a cluster similar to the previous one, except all of the workloads were on a single E2 node with 16 CPU and 64 GB of memory ("e2 combined" on the graph). The parse time gains here were fairly substantial, with the runs coming in bi-modally at either **360ms** or **410ms**, a 15%-25% improvement over even the best-case for separating out the E2 nodes.

I don't currently have a good explanation for the variability in parse times when they're all on the same machine. Maybe proximity to the database node?

Does using faster CPUs noticeably improve parse time?

tl;dr: YES, by around 10-20%, with some variation

The E2 machines are fairly low-power compared to some of the alternatives. We tried setting up the same experiment using N2D machines, which have much higher scores in CPU benchmarks. These were much more consistent ("n2d combined") and gave an average parse time of about **330ms**, about a 10% improvement over the best-case from E2, and without any bimodal distribution.

We also tried using T2D machines, which are even more powerful, although currently not priced in a way that we are likely able to use. These had an average parse time of **320ms**, a 2-3% improvement.

Does using Compact Placement for nodes improve parse time?

tl;dr: sometimes, probably depending on whether it's on the same physical machine

Compact placement is a way to specify that you'd like your nodes to be as close to each other as possible, to reduce latency. This would be a decent option if for some reason we wanted to keep everything on separate nodes, but still wanted some of the latency benefits.

Compact placement is not available for E2 nodes, but it is available for N2D. We ran tests with this configuration ("n2d compact") and got a bimodal distribution, either matching almost exactly the latency we saw from "n2d separate", or matching the latency we saw from "n2d combined". My

guess is that the improvement occurred when the compact placement policy allowed the separate nodes to be scheduled on the same physical machine.

Does using 1 thread per core improve latency vs 2 threads per core?

tl;dr: no

At this point, we're closing in on the idea that "N2D combined" is probably roughly the architecture we're going to prioritize. There is an option to have either 2 threads per CPU core, or 1 thread per CPU core. 2 is the default, but 1 could be useful if there are CPU-heavy workloads that don't need to context-switch very often.

We tried running the same "N2D combined" experiment with 1 thread per core, and saw no difference whatsoever. Probably not worth further investigation.

What is the resource bottleneck if we want to run more jobrunners?

tl;dr: it's the database, stupid!

We can scale up the number of jobrunners almost infinitely when there's work to do, but we need to be mindful of the other resources that get consumed during job runs. These are mainly the database CPU, redis-main-cache CPU, redis-mediawiki (parser cache) CPU and i/o, and elasticsearch CPU.

Normally the wiki runs up to 11 jobrunners at 1 CPU each, which is enough to finish rs.wiki Grand Exchange updates in about 50 minutes, including scale-up. But for these tests we used 8 jobrunners at 4 CPU each, or about triple the normal rate. Under this scenario, after a few runs:

- database CPU utilization peaked at about 78-81% of 2 vCPU
- after accounting for the rdb saving process, redis-main-cache peaked at 46% of 1 vCPU
- after accounting for the rdb saving process, redis-mediawiki peaked at 8% of 1 vCPU
- elasticsearch peaked at 35% of 1 vCPU

It's fairly clear from these tests that the database is the main bottleneck for scaling up the jobrunners. We also tried running the same tests with only 4 jobrunners at 4 CPU, and the database peaked at about 45%. Since this is a bit more than half of what it peaked at with twice as many jobrunners, one might surmise that even at 81%, it is starting to throttle a little when 32 CPU worth of jobrunners are hitting it.

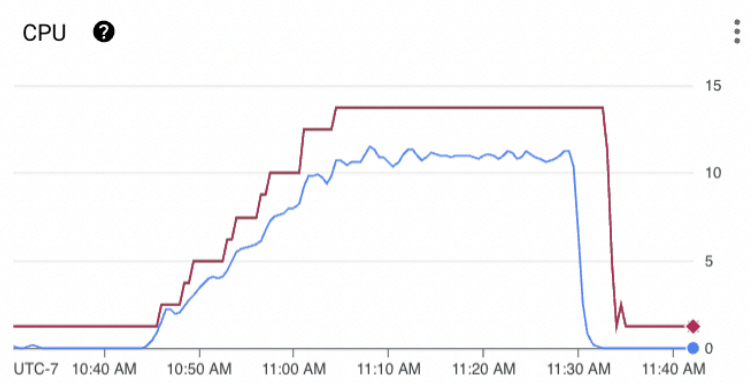
I think the move here is to limit the jobrunner to 6x4 CPU, to account for the various other things that could command database resources (mainly the maintenance cron jobs, that can take about 30% once a day for regenerating special pages). Generally the database handles maxed-out-CPU a bit better than the mediawiki pods do, but it's still good to try to avoid going too high.

If we want to consider improving this in the future so that we can run even more jobrunners, our two options are to either scale up the number of CPU on the database (expensive!) or remove Semantic MediaWiki, since that's responsible for about 75% of all writes and 50% of all reads on the site.

How fast can jobrunners scale up in various configurations?

tl;dr: about 2 minutes per new node, regardless of other considerations

Currently rs.wiki Grand Exchange updates take 45-50 minutes from start to finish, but the resource usage is sort of trapezoidally shaped:



Because we scale up using a horizontal pod autoscaler based on CPU utilization, it doesn't immediately use as many pods as it can, but rather scales up one at a time, somewhat non-linearly. In our current production, it seems like every additional pod (node) scaled up takes about 2 or 3 minutes.

We tried doing horizontal scaling on a node that already had plenty of free capacity, and found that it scheduled 7 additional 4-CPU pods almost instantly (within 2-3 minutes). When we instead used a node pool that provisioned new nodes as needed, we were back to about 2 minutes per new node.

Notably, this doesn't really depend on the size of the nodes/pods – it takes the same amount of time to provision a new 1-CPU pod as it does to provision a new 4-CPU pod. So in general we want to use bigger pods so the scale-up is quicker.

Spot VMs are extraordinarily cheap when not on autopilot – a 4-CPU N2D costs about the same hourly as a single (worse) CPU on autopilot. Cost is a really low-order concern for jobrunner decisions.

The proposal is to use **4-CPU jobrunner pods up to 6 maximum**, which would move the average rs.wiki Grand Exchange update from 45-50 minutes down to approximately 20.

There are additional things we could do to make the scale-up faster, either by using custom metrics related to job counts, or by using an overflow pool with a much larger node (say, a default of one jobrunner with 4 CPU, but schedule the rest on an overflow node with 20 CPU). By my calculations, neither of these would improve the total Grand Exchange update time by more than about 20%. If we want to significantly reduce this further, we'd need to either:

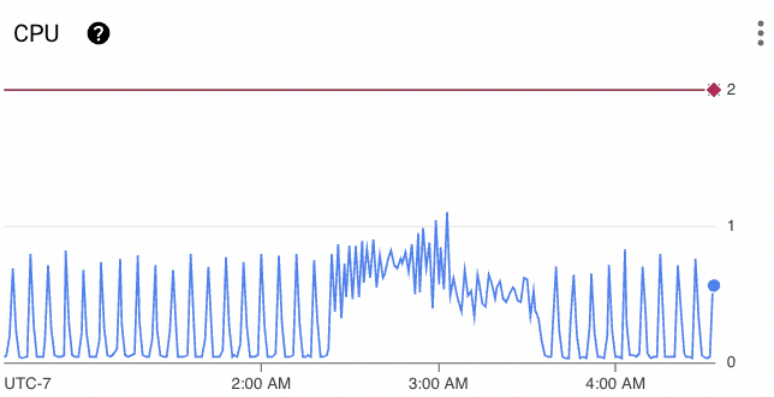
- Reduce the number of pages touched by a Grand Exchange update (a la my ramblings)
- Increase database CPU (probably not worth it due to cost)
- Get rid of Semantic MediaWiki

Should we change redis backup frequency?

tl;dr: probably change it to once every 3 hours?

By default, redis periodically saves its on-memory data to disk in an rdb file. This happens once every 5 minutes normally, but may happen up to once a minute if many writes happen at once. The way it does this is it forks a process that writes the entire content of the database to an LZF-compressed file. This can take about a minute when the database is ~8-10 GB in memory, and can consume nearly a whole CPU core during the process. Then, if the database needs to be restarted for whatever reason, it reads the contents of the rdb file into memory.

This is a bit overkill for us, since the data we store on redis is not so mission-critical – it's primarily a cache for other precomputed data and database entries. If we lose a little bit of redis data, nothing bad really happens. So having a full CPU (really two, because we have two big redis instances) almost constantly performing backups that we almost never need, and even if we do need, are fine being rather stale…seems a bit unnecessary, when those CPU could, in high-usage moments, be used for more pressing tasks.



redis-main-cache CPU during job run – this is mostly the backup process

I would propose that we reduce the frequency of rdb backups to once every 3 hours, maybe slightly offset so the parser cache and main cache are not likely to be backing up at the same time. Alternatively, if there's some way to lower the priority of these jobs, or limit their CPU, that is probably worth exploring. The backup process only takes about 60 seconds so slowing it down slightly isn't really a problem, although we don't want it to take interminably long, because the whole thing relies on "copy-on-write" memory semantics, and leaving the process running for too long could result in too many memory pages being copied, causing memory pressure issues for us.

I am also open to just not running the backup process at all, but this could have some negative consequences if we need to start with a completely dead parser cache.

How much extra CPU/memory do we need to do rolling deployments?

tl;dr: nothing for the StatefulSets, one pod worth for everything else

Be aware that the scheduling constraints here involve Kubernetes *requests*, rather than *limits*. The question here is basically how much additional CPU/memory we need to keep un-requested to be able to do rolling deployments.

For StatefulSets, we really aren't doing rolling deployments to begin with (there's only one of each). So even if the main node we're using has almost no un-requested memory to allocate, we can replace the existing redis/elasticsearch StatefulSets.

This is in contrast to the actual deployments (mainly the mediawiki pods), which can't be replaced unless there's enough un-requested CPU/memory to temporarily schedule a new one.

To illustrate, imagine we have mediawiki pods that request 3 CPU and 3 GB of memory. Normally we have two of them, but we want to deploy a new image. If we only have 2 CPU (or 2 GB of memory) un-requested, it won't be able to update them at all. If we have (say) 3.5 CPU and 3.5 GB of memory un-requested, that's enough to replace them one at a time, and the deployment will succeed.

mediawiki pods are going to be the most CPU-hungry things on our main node – probably we'll have 2 pods each allowed to use up to 4 CPU. But it seems silly to need to leave 4 CPU unallocated on the node just so we can do the rolling deployment properly.

I think the solution here is to set the requests significantly lower than the limits. This might be a bit of an anti-pattern for Kubernetes, but we know exactly which pods we want to have on each node, and requests are really only used to help Kubernetes decide where to place things. This way we can set the CPU limits for the mediawiki pods to (say) 4 each, but keep the requests lower (perhaps even zero, if we have other rules in place to prevent too many mediawiki pods from being scheduled in one place) so it's possible to do the full rolling deployment.

What resources does k8s need on the nodes just for boilerplate?

For a 16 CPU, 64 GB machine that is the only node in the cluster:

- 0.1 CPU taken, 0.7 CPU used for other pods
- 2.5G memory taken, ~0.7G used for other pods

This gets significantly worse if we have lots of nodes at once. Another reason to consider fewer nodes (perhaps as few as one for everything except jobrunner and mediawiki overflow).

For 4 CPU, 4 GB spot nodes:

- 0.08 CPU taken, 0.31 CPU used for other pods
- 1.05G memory taken, 0.39G used for other pods

Are we vulnerable to downtime or bad pod placement if k8s upgrades happen?

tl;dr: yes, but probably not any more with this new architecture than currently

We are looking at somewhere between 1-5 minutes of StatefulSet downtime whenever nodes get updated to the latest Kubernetes version, which happens about once a month. This already happens (see error logs for August 4th, for example), and as far as I can tell, would not get any worse with this new architecture, so long as we have the "surge" setting turned on that will temporarily overprovision nodes during upgrades.

I was concerned at first that this meant we might end up with our pods skewed across multiple nodes (causing latency problems) when all is said and done, but from testing this does not seem to be the case.

Should we disable Kubernetes node auto-upgrades?

My hot take is that until we have some way to make the site not have degraded functionality when the StatefulSets are down, the GKE version auto-upgrades do more harm than good. They currently happen somewhere between once and twice a month, but if we stop being in the release channel, we can get away with upgrading as infrequently as once per 9 months (per docs, nodes can be up to 2 minor versions away from the control plane version before they start getting auto-upgraded).

Can we make deployments less catastrophic when components are missing?

We have degraded functionality when the StatefulSets are down due to either redeployments, or Kubernetes version upgrades. Two things to consider here:

- Right now pretty much any request to the wiki will fail with a LogicException in the message cache if redis-main-cache is down. I think we can safely remove this exception, although the downside is that MediaWiki messages (most notably the sidebar) won't load properly if it's down. I think this is worth it when the alternative is not being able to load anything. However, Kitty brings up a good point that we should not cache these partially-successful pageviews, suggesting `RequestContext::getMain()->getOutput()->disableClientCache();` as a replacement for the LogicException. Note that even if we do this, searching will still fail somewhat spectacularly if elasticsearch is missing. Maybe there's something we can do to redirect to Google site search when we're down?
- What can we do to make redis/elasticsearch redundant so they don't just disappear during deployments/upgrades?

Can create "overflow" mediawiki pods on spot instances that only run if the HPA hits a CPU threshold?

tl;dr: seems like it

Even though we generally want the mediawiki pods to be on the same node as everything else for latency reasons, there will be times where we need to scale up in response to overwhelming utilization.

Since my htmlCacheUpdate changes, we no longer have spiky webserver load from Grand Exchange updates or other high-use template purges, and most of the CPU spikes in the last month

are from badly-behaving crawlers. We can't predict when these will happen, so we need to be able to scale up automatically if the mediawiki pods on the main node get overwhelmed. It's okay if these ones are a bit slower than normal ones, since they will only be active in rare cases.

Kitty came up with a clever way to put additional mediawiki pods on spot instances in an "overflow" pool that I still need to test.

What happens if no spot instances are available?

Not sure. Can we back off to using non-spot instances if needed? Does this happen automagically?

Sizing

If we go with N2D, we basically need to use either a 12-CPU machine or a 16-CPU machine for the main node, due to GCP constraints. I think we can fit it all on a 12-CPU machine without much issue.

The main guiding factors here:

- Memory is way cheaper and more flexible than CPU
- It's okay to maybe slightly oversubscribe CPU, but never memory
- No mediawiki pod (4 CPU) has ever been above 1.3 GB of memory in the last 6 weeks
- No mediawiki-jobrunner pod (1 CPU) has ever been over 600 MB of memory in the last 6 weeks

pod	placement	count	cpu_limit	mem_limit	cpu_req	mem_req
mediawiki	default + spot	2-5	4	2.5	0	0
mediawiki-logging-sidecar	sidecar w/ mediawiki	varies	0.25	0.5	0	0
mediawiki-jobrunner	spot	1-6	4	2.5	3	2.5
redis-main-cache	default	1	2	13	1	13
redis-mediawiki	default	1	2	13	1	13
redis-job-queue	default	1	0.5	0.5	0.1	0.5
elasticsearch-mwsearch	default	1	1.5	7.5	1	7.5
gcsproxy	default	1	0.5	0.2	0.1	0.2
jobchron	default	1	0.05	0.2	0.05	0.2
nginx	default	2	0.25	0.2	0.05	0.2
shellbox-syntaxhighlight	default	1	0.5	0.2	0.1	0.2
shellbox-timeline	default	1	0.01	0.2	0.01	0.2

Open questions about sizing:

- What should we set mediawiki requests at? We want it to be low enough that we can do rolling deployments without a bunch of wasted resources, but also don't want the overflow nodes to schedule more than one per 4/4 node, I think. Or maybe it's not actually a problem to schedule 2 pods per 4/4, since it will never really use enough memory to hit eviction limits, and CPU throttling isn't a huge issue.
- What's the reason we have 2 nginx pods? Does this help with deployments?

Pricing and committed use discounts

resource	count	price (normal)	price (1 yr)	price (3 yr)
N2D CPU (no SUD)	12	22.10294	13.92	9.95
N2D RAM (no SUD)	48	2.96234	1.87	1.33
CloudSQL HA CPU	2	66.284	49.713	31.8134
CloudSQL HA RAM	13	11.242	8.4315	5.3947
N2D CPU (spot)	5	2.23	2.23	2.23
N2D RAM (spot)	5	0.3	0.3	0.3
TOTAL		**698.7916**	**478.4855**	**329.6479**

Currently we spend about $806 a month on CPU/memory for production Kubernetes nodes, plus $280 a month on CPU/memory for the database.

Using on-demand machines in the newly proposed architecture (12 CPU, 48 GB memory, plus overflow), we estimate the Kubernetes cost would decline to about $420 per month, representing a yearly savings of about $4600, in addition to any reduction in dev cluster costs.

Additionally, Google Cloud has a pricing mechanism called committed use discounts where if we commit to a certain set of resources for a year (or three years), we get steep price discounts. We can use this for both the Kubernetes nodes and the CloudSQL databases, and 1-year committed use discounts for our proposed architecture would save an additional $2600 per year. 3-year committed use discounts would save an additional $1800 per year on top of this, but I think that's too long to commit to specific machines and sizes.

The downside is that if we decide we don't like the specific machines, or location, or count, we're pretty much stuck with this particular infrastructure for a year (or a $6000 sunk cost). Note that if we have to scale UP, this isn't really a problem, since we can just use more on-demand resources as needed. It's only an issue if we decide we need to scale down (which really doesn't seem necessary at that new price point, considering ~9 months ago we were spending 4 times this much on the same resources).

I think we should spend about a month using the on-demand pricing just to make sure this particular architecture works and is sized appropriately. After that we can purchase the 1-year CUDs – it's not an upfront cost, but rather a contract where we commit to paying however much every month.

Decisions

- **Put everything except jobrunners on one 12-CPU, ~48 GB machine**
- **jobrunners should be 4/4 on spot-pool N2D nodes scaling up to 6 total**
- **mediawiki pods should overflow to spot-pool as well - 3 max?**
- **cronjobs to be scheduled on same spot-pool as jobrunners**
- **Change redis "save" configuration either not run at all, or run every few hours**
- **Remove LogicException in MessageCache in mediawiki core, replace with uncacheable**
- **Purchase 1-year committed-use discount for 12/48(?) N2D CPU/memory**
- **Purchase 1-year committed-use discount for CloudSQL 2/13 HA**

Deployment plan

This will require some read-only time, but likely no downtime. We want to do this during a low-activity time (but probably not coinciding with game maintenance, since that often precedes important updates)

1. Make main-prod db read-only
2. Clone disks for all 4 StatefulSets (a few minutes)
3. Deploy new cluster (5-10 minutes)
4. Can we confirm this works at some level before switching over the load balancer?
5. Switch over load balancer to new IP
6. Do we need to delete the old cluster immediately, or can we keep it around if we need to rollback?

Further performance work

Out of scope for this project but fun to think about:

- Eliminate Semantic MediaWiki and replace with Bucket
- Experiment with LuaJIT as Scribunto backend
- Make LinkBatch support file links as well
- Explore other i/o batching
- Switch to PHP 8.0 or 8.1 - some compatibility issues with Semantic MediaWiki still, but might be another 20-30% speedup
- Explore Kartographer slowness, including JSON verification

Final thoughts

- Is it a dumb idea to put everything on one machine? In theory, we are trading higher redundancy and reliability (in the case of zonal outages) for better latency. Is this gonna come back to bite us in the ass?
- This new architecture feels kind of like an anti-pattern for Kubernetes. Is this a sign that maybe we don't want to be using Kubernetes in the long term? I like how we can automatically scale resources up and down with demand, and do rolling deployments on all of the components, but everything else in Kubernetes-land feels like overkill for our use case, artificially increasing resource requirements and (most likely, although hard to test) increasing latency on basic operations. I'm not sure what else is out there that would be appropriate for autoscaling that is a bit less heavy-duty, although I'm interested in trying out various ideas. Note that whatever CUDs we buy would not necessarily force us to use Kubernetes for the full year – they would just be commitments to specific machine types.

Sign off

- [ ] Cook
- [ ] Kitty
- [ ] Jayden
- [ ] Gaz