

How Netflix Provisions Optimal Cloud Deployments of Cassandra

Joey Lynch
Senior Software Engineer
Cloud Data Engineering - Netflix

Speaker

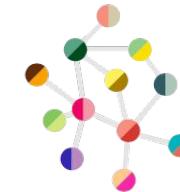
Joey Lynch



<https://jolynch.github.io/>

Senior Software Engineer
Cloud Data Engineering at Netflix

Database shepherd and data wrangler



Show me the
Code!

Service Capacity Modeling



A generic toolkit for modeling capacity requirements in the cloud. Pricing information included in this repository are public prices.

<https://github.com/Netflix-Skunkworks/service-capacity-modeling>

Outline

Understanding Hardware

Computers are shaped differently
Computers cost money

Capacity Planning

Requirement Language
Capacity Planning - Queues oh my
Cassandra Capacity Planning Model

Monitoring your Choices

Key Capacity Metrics to Monitor

Capacity Planning 101

$$M(D, H, PL) \rightarrow C$$

M = Workload Capacity Model

D = User Desire

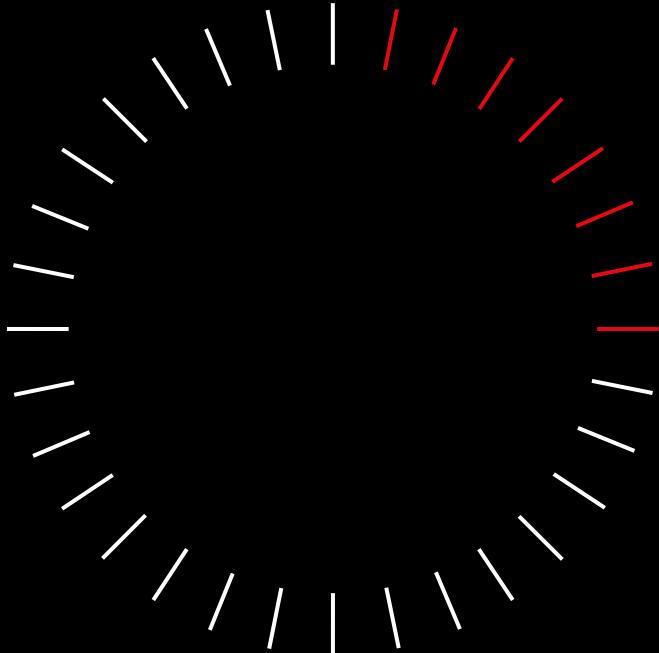
H = Hardware Profile

PL = Current Pricing and Lifecycle

C = Candidate Cluster

Under- standing Hardware

There are a lot of computers
... and they cost money



Capacity Planning 101

$$M(D, H, PL) \rightarrow C$$

M = Workload Capacity Model

D = User Desire

H = Hardware Profile

PL = Current Pricing and Lifecycle

C = Candidate Cluster

Hardware

Amazon EC2 Instance Comparison										
Instance Type	CPU		Memory		Storage		Networking		Compute Performance	
	Cores	Threads	GB	GB	Volume Type	IOPS	Bandwidth	Mbps	Latency	Latency
t1.micro	1	2	0.75	0.75	Standard	0	1 Gbps	100	1 ms	1 ms
t1.small	2	4	1.75	1.75	Standard	0	1 Gbps	100	1 ms	1 ms
t1.medium	4	8	3.5	3.5	Standard	0	1 Gbps	100	1 ms	1 ms
t1.large	8	16	7	7	Standard	0	1 Gbps	100	1 ms	1 ms
t1.xlarge	16	32	14	14	Standard	0	1 Gbps	100	1 ms	1 ms
t1.2xlarge	32	64	28	28	Standard	0	1 Gbps	100	1 ms	1 ms
t1.4xlarge	64	128	56	56	Standard	0	1 Gbps	100	1 ms	1 ms
t1.6xlarge	96	192	84	84	Standard	0	1 Gbps	100	1 ms	1 ms
t1.8xlarge	128	256	112	112	Standard	0	1 Gbps	100	1 ms	1 ms
t1.10xlarge	160	320	140	140	Standard	0	1 Gbps	100	1 ms	1 ms
t1.12xlarge	192	384	178	178	Standard	0	1 Gbps	100	1 ms	1 ms
t1.14xlarge	224	448	216	216	Standard	0	1 Gbps	100	1 ms	1 ms
t1.16xlarge	256	512	254	254	Standard	0	1 Gbps	100	1 ms	1 ms
t1.18xlarge	288	576	292	292	Standard	0	1 Gbps	100	1 ms	1 ms
t1.20xlarge	320	640	330	330	Standard	0	1 Gbps	100	1 ms	1 ms
t1.22xlarge	352	704	368	368	Standard	0	1 Gbps	100	1 ms	1 ms
t1.24xlarge	384	768	406	406	Standard	0	1 Gbps	100	1 ms	1 ms
t1.26xlarge	416	832	444	444	Standard	0	1 Gbps	100	1 ms	1 ms
t1.28xlarge	448	896	482	482	Standard	0	1 Gbps	100	1 ms	1 ms
t1.30xlarge	480	960	520	520	Standard	0	1 Gbps	100	1 ms	1 ms
t1.32xlarge	512	1024	558	558	Standard	0	1 Gbps	100	1 ms	1 ms
t1.34xlarge	544	1088	596	596	Standard	0	1 Gbps	100	1 ms	1 ms
t1.36xlarge	576	1152	634	634	Standard	0	1 Gbps	100	1 ms	1 ms
t1.38xlarge	608	1216	672	672	Standard	0	1 Gbps	100	1 ms	1 ms
t1.40xlarge	640	1280	710	710	Standard	0	1 Gbps	100	1 ms	1 ms
t1.42xlarge	672	1344	748	748	Standard	0	1 Gbps	100	1 ms	1 ms
t1.44xlarge	704	1408	786	786	Standard	0	1 Gbps	100	1 ms	1 ms
t1.46xlarge	736	1472	824	824	Standard	0	1 Gbps	100	1 ms	1 ms
t1.48xlarge	768	1536	862	862	Standard	0	1 Gbps	100	1 ms	1 ms
t1.50xlarge	800	1600	900	900	Standard	0	1 Gbps	100	1 ms	1 ms
t1.52xlarge	832	1664	938	938	Standard	0	1 Gbps	100	1 ms	1 ms
t1.54xlarge	864	1728	976	976	Standard	0	1 Gbps	100	1 ms	1 ms
t1.56xlarge	904	1792	1014	1014	Standard	0	1 Gbps	100	1 ms	1 ms
t1.58xlarge	936	1856	1052	1052	Standard	0	1 Gbps	100	1 ms	1 ms
t1.60xlarge	968	1920	1090	1090	Standard	0	1 Gbps	100	1 ms	1 ms
t1.62xlarge	1000	1984	1128	1128	Standard	0	1 Gbps	100	1 ms	1 ms
t1.64xlarge	1032	2048	1166	1166	Standard	0	1 Gbps	100	1 ms	1 ms
t1.66xlarge	1064	2112	1204	1204	Standard	0	1 Gbps	100	1 ms	1 ms
t1.68xlarge	1104	2176	1242	1242	Standard	0	1 Gbps	100	1 ms	1 ms
t1.70xlarge	1136	2240	1280	1280	Standard	0	1 Gbps	100	1 ms	1 ms
t1.72xlarge	1168	2304	1318	1318	Standard	0	1 Gbps	100	1 ms	1 ms
t1.74xlarge	1200	2368	1356	1356	Standard	0	1 Gbps	100	1 ms	1 ms
t1.76xlarge	1232	2432	1394	1394	Standard	0	1 Gbps	100	1 ms	1 ms
t1.78xlarge	1264	2496	1432	1432	Standard	0	1 Gbps	100	1 ms	1 ms
t1.80xlarge	1300	2560	1470	1470	Standard	0	1 Gbps	100	1 ms	1 ms
t1.82xlarge	1332	2624	1508	1508	Standard	0	1 Gbps	100	1 ms	1 ms
t1.84xlarge	1364	2688	1546	1546	Standard	0	1 Gbps	100	1 ms	1 ms
t1.86xlarge	1400	2752	1584	1584	Standard	0	1 Gbps	100	1 ms	1 ms
t1.88xlarge	1432	2816	1622	1622	Standard	0	1 Gbps	100	1 ms	1 ms
t1.90xlarge	1464	2880	1660	1660	Standard	0	1 Gbps	100	1 ms	1 ms
t1.92xlarge	1500	2944	1708	1708	Standard	0	1 Gbps	100	1 ms	1 ms
t1.94xlarge	1532	3008	1746	1746	Standard	0	1 Gbps	100	1 ms	1 ms
t1.96xlarge	1564	3072	1784	1784	Standard	0	1 Gbps	100	1 ms	1 ms
t1.98xlarge	1600	3136	1822	1822	Standard	0	1 Gbps	100	1 ms	1 ms
t1.100xlarge	1632	3200	1860	1860	Standard	0	1 Gbps	100	1 ms	1 ms
t1.102xlarge	1664	3264	1908	1908	Standard	0	1 Gbps	100	1 ms	1 ms
t1.104xlarge	1700	3328	1946	1946	Standard	0	1 Gbps	100	1 ms	1 ms
t1.106xlarge	1732	3392	1984	1984	Standard	0	1 Gbps	100	1 ms	1 ms
t1.108xlarge	1764	3456	2022	2022	Standard	0	1 Gbps	100	1 ms	1 ms
t1.110xlarge	1800	3520	2060	2060	Standard	0	1 Gbps	100	1 ms	1 ms
t1.112xlarge	1832	3584	2108	2108	Standard	0	1 Gbps	100	1 ms	1 ms
t1.114xlarge	1864	3648	2146	2146	Standard	0	1 Gbps	100	1 ms	1 ms
t1.116xlarge	1900	3712	2184	2184	Standard	0	1 Gbps	100	1 ms	1 ms
t1.118xlarge	1932	3776	2222	2222	Standard	0	1 Gbps	100	1 ms	1 ms
t1.120xlarge	1964	3840	2260	2260	Standard	0	1 Gbps	100	1 ms	1 ms
t1.122xlarge	2000	3904	2308	2308	Standard	0	1 Gbps	100	1 ms	1 ms
t1.124xlarge	2032	3968	2346	2346	Standard	0	1 Gbps	100	1 ms	1 ms
t1.126xlarge	2064	4032	2384	2384	Standard	0	1 Gbps	100	1 ms	1 ms
t1.128xlarge	2100	4096	2422	2422	Standard	0	1 Gbps	100	1 ms	1 ms
t1.130xlarge	2132	4160	2460	2460	Standard	0	1 Gbps	100	1 ms	1 ms
t1.132xlarge	2164	4224	2508	2508	Standard	0	1 Gbps	100	1 ms	1 ms
t1.134xlarge	2200	4288	2546	2546	Standard	0	1 Gbps	100	1 ms	1 ms
t1.136xlarge	2232	4352	2584	2584	Standard	0	1 Gbps	100	1 ms	1 ms
t1.138xlarge	2264	4416	2622	2622	Standard	0	1 Gbps	100	1 ms	1 ms
t1.140xlarge	2300	4480	2660	2660	Standard	0	1 Gbps	100	1 ms	1 ms
t1.142xlarge	2332	4544	2708	2708	Standard	0	1 Gbps	100	1 ms	1 ms
t1.144xlarge	2364	4608	2746	2746	Standard	0	1 Gbps	100	1 ms	1 ms
t1.146xlarge	2400	4672	2784	2784	Standard	0	1 Gbps	100	1 ms	1 ms
t1.148xlarge	2432	4736	2822	2822	Standard	0	1 Gbps	100	1 ms	1 ms
t1.150xlarge	2464	4800	2860	2860	Standard	0	1 Gbps	100	1 ms	1 ms
t1.152xlarge	2500	4864	2908	2908	Standard	0	1 Gbps	100	1 ms	1 ms
t1.154xlarge	2532	4928	2946	2946	Standard	0	1 Gbps	100	1 ms	1 ms
t1.156xlarge	2564	4992	2984	2984	Standard	0	1 Gbps	100	1 ms	1 ms
t1.158xlarge	2600	5056	3022	3022	Standard	0	1 Gbps	100	1 ms	1 ms
t1.160xlarge	2632	5120	3060	3060	Standard	0	1 Gbps	100	1 ms	1 ms
t1.162xlarge	2664	5184	3108	3108	Standard	0	1 Gbps	100	1 ms	1 ms
t1.164xlarge	2700	5248	3146	3146	Standard	0	1 Gbps	100	1 ms	1 ms
t1.166xlarge	2732	5312	3184	3184	Standard	0	1 Gbps	100	1 ms	1 ms
t1.168xlarge	2764	5376	3222	3222	Standard	0	1 Gbps	100	1 ms	1 ms
t1.170xlarge	2800	5440	3260	3260	Standard	0	1 Gbps	100	1 ms	1 ms
t1.172xlarge	2832	5504	3308	3308	Standard	0	1 Gbps	100	1 ms	1 ms
t1.174xlarge	2864	5568	3346	3346	Standard	0	1 Gbps	100	1 ms	1 ms
t1.176xlarge	2900	5632	3384	3384	Standard	0	1 Gbps	100	1 ms	1 ms
t1.178xlarge	2932	5696	3422	3422	Standard	0	1 Gbps	100	1 ms	1 ms
t1.180xlarge	2964	5760	3460	3460	Standard	0	1 Gbps	100	1 ms	1 ms
t1.182xlarge	3000	5824	3508	3508	Standard	0	1 Gbps	100	1 ms	1 ms
t1.184xlarge	3032	5888	3546	3546	Standard	0	1 Gbps	100	1 ms	1 ms
t1.186xlarge	3064	5952	3584	3584	Standard	0	1 Gbps	100	1 ms	1 ms
t1.188xlarge	3100	6016	3622	3622	Standard	0	1 Gbps	100	1 ms	1 ms
t1.190xlarge	3132	6080	3660	3660	Standard	0	1 Gbps	100	1 ms	1 ms
t1.192xlarge	3164	6144	3708	3708	Standard	0	1 Gbps	100	1 ms	1 ms
t1.194xlarge	3200	6208	3746	3746	Standard	0	1 Gbps	100	1 ms	1 ms
t1.196xlarge	3232	6272	3784	3784	Standard	0	1 Gbps	100	1 ms	1 ms
t1.198xlarge	3264	6336	3822	3822	Standard	0	1 Gbps	100	1 ms	1 ms
t1.200xlarge	3300	6400	3860	3860	Standard	0	1 Gbps	100	1 ms	1 ms
t1.202xlarge	3332	6464	3908	3908	Standard	0	1 Gbps	100	1 ms	1 ms
t1.204xlarge	3364	6528	3946	3946	Standard	0	1 Gbps	100	1 ms	1 ms
t1.206xlarge	3400	6592	3984	3984	Standard	0	1 Gbps	100	1 ms	1 ms
t1.208xlarge	3432	6656	4022	4022	Standard	0	1 Gbps	100	1 ms	1 ms
t1.210xlarge	3464	6720	4060	4060	Standard	0	1 Gbps	100	1 ms	1 ms
t1.212xlarge	3500	6784	4108	4108	Standard	0	1 Gbps	100	1 ms	1 ms
t1.214xlarge	3532	6848	4146	4146	Standard	0	1 Gbps	100	1 ms	1 ms
t1.216xlarge	3564	6912	4184	4184	Standard	0	1 Gbps	100	1 ms	1 ms
t1.218xlarge	3600	6976	4222	4222	Standard	0	1 Gbps	100	1 ms	1 ms
t1.220xlarge	3632	7040	4260	4260	Standard	0	1 Gbps	100	1 ms	1 ms
t1.222xlarge	3664	7104	4308	4308	Standard	0	1 Gbps	100	1 ms	1 ms
t1.224xlarge	3700	7168	4346	4346	Standard	0	1 Gbps	100	1 ms	1 ms
t1.226xlarge	3732	7232	4384	4384	Standard	0	1 Gbps	100	1 ms	1 ms
t1.228xlarge	3764	7296	4422	4422	Standard	0	1 Gbps	100	1 ms	1 ms
t1.230xlarge	3800	7360	4460	4460	Standard	0	1 Gbps	100	1 ms	1 ms
t1.232xlarge	3832	7424	4508	4508	Standard	0	1 Gbps	100	1 ms	1 ms
t1.234xlarge	3864	7488	4546	4546	Standard	0	1 Gbps	100	1 ms	1 ms
t1.236xlarge	3900	7552	4584	4584	Standard	0	1 Gbps	100	1 ms	1 ms
t1.238xlarge	3932	7616	4622	4622	Standard	0	1 Gbps	100	1 ms	1 ms
t1.240xlarge	3964	7680	4660	4660	Standard	0	1 Gbps	100	1 ms	1 ms
t1.242xlarge	4000	7744	4708	4708	Standard	0	1 Gbps	100	1 ms	1 ms
t1.244xlarge	4032	7808	4746	4746	Standard	0	1 Gbps	1		

Hundreds of choices

With confusing names

No indication of lifecycle (alpha, beta, stable, deprecated)

Hardware

Region: US East (N. Virginia) ▾ Pricing Unit: Instance ▾ Cost: Annually ▾ Reserved: 3-year - Full Upfront ▾ Columns ▾ Compare Selected ▾ Clear Filters ▾ CSV

Filter: Min Memory (GiB): 0 Min vCPUs: 0 Min Memory/vCPU (GiB/vCPU): 0

Name API Name

Search m5d.2x

M5 General Purpose Double Extra Large m5d.2xlarge

Different Prices Why? Because it's frustrating to compare instances using Amazon's own instance Who? It was started by @powdahound, contributed to by many, is now managed by How? Data is scraped from multiple pages on the AWS site. This was last done at Warning: This site is not maintained by or affiliated with Amazon. The data shown

1-year - No Upfront
1-year - Partial Upfront
1-year - Full Upfront
3-year - No Upfront
3-year - Partial Upfront
3-year - Full Upfront

32.0 GiB 8 vCPUs 300 GiB NVMe SSD Up to 10 Gigabit \$3959.520000 annually \$1488.665640 annually \$1899.168000 annually

your improvements on GitHub.
Not Entirely Accurate
Please report issues you see.

This changes every minute

Relevant information to the choice **changes rapidly** and **is not always accurate**.

Problem

$$M(D, H, PL) \rightarrow C$$

M = Workload Capacity Model

D = User Desire

H = Hardware Profile

PL = Current Pricing and Lifecycle

C = Candidate Cluster

We do not have accurate hardware profiles

We do not know company specific pricing and lifecycle information

Solution?

Find the instance type labeled "database class" and buy that

Solution?

Find the instance type labeled "database class" and buy that

Search for conference talks by "big users" and use whatever they use.

Solution?

Find the instance type labeled "database class" and buy that

Search for conference talks by "big users" and use whatever they use.

We can do better

Hardware

Capacity: How much CPU, RAM, Network, Disk?

Latency: How fast are the CPUs, NICs, and Drives?

Lifecycle: Is this alpha or stable?

Price: How much do I pay?

Hardware

Capacity: How much CPU, RAM, Network, Disk?
You can measure these

Latency: How fast are the CPUs, NICs, and Drives?

Lifecycle: Is this alpha or stable?
This depends on your deployment

Price: How much do I pay?

Hardware Lifecycle

Would friends let friends launch on m3 instances?

Does your software stack work on arm64?

Hardware Lifecycle

At Netflix

Alpha: Hardware preview (m6g)

Beta: Production testing (r5dn)

Stable: Use in production (m5)

Deprecated: Stop using (i3 -> i3en)

End-of-life: Do not use (m3, i2, ...)

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/interface.py#L116-L131

Solution!

We can know!

Enumerate Hardware Shapes
Measure their performance

https://github.com/Netflix-Skunkworks/service-capacity-modeling/tree/main/service_capacity_modeling/hardware/profiles

Solution!

We can know!

Enumerate Hardware Shapes
Measure their performance

Enrich with context!

Layer on pricing and lifecycle

https://github.com/Netflix-Skunkworks/service-capacity-modeling/tree/main/service_capacity_modeling/hardware/profiles

Solution!

How do we measure?

Generate Load ([iperf](#), [ndbench](#),
[netperf](#), [fio](#))

Measure ([bcc](#), metrics, etc..)

How do we price?

Layer company [pricing on top](#) of
shape definitions

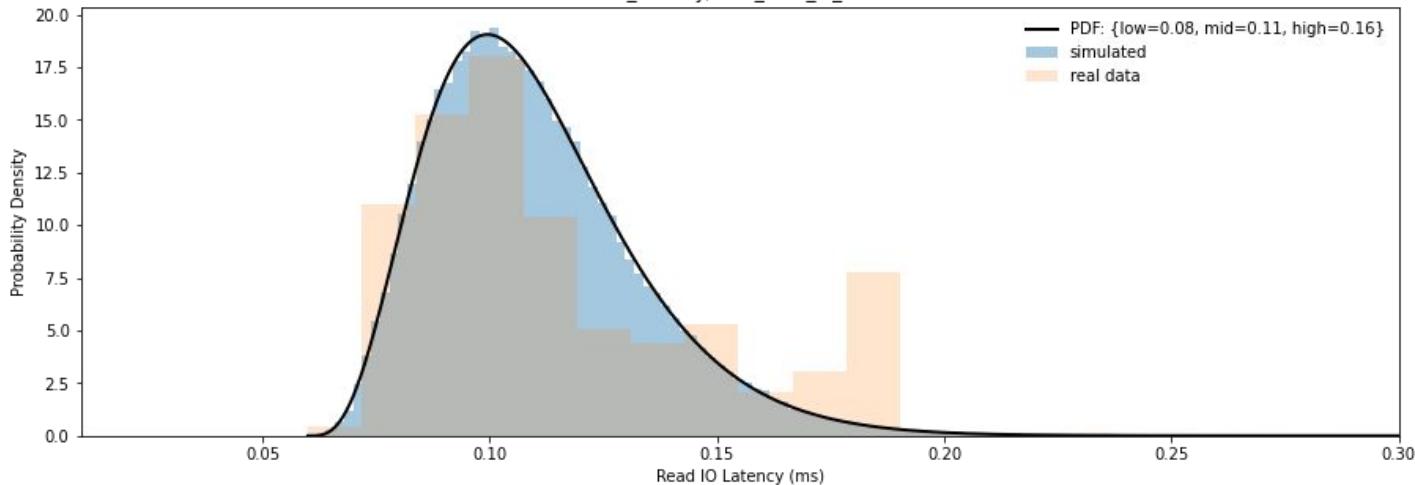
Solution!

For a concrete example let's model a m5d drive which we `model` with an io latency distribution. Data for comparision comes from using `biosnoop` and `histogram.py` on a Cassandra server (the threads that are servicing reads are from the SharedPool).

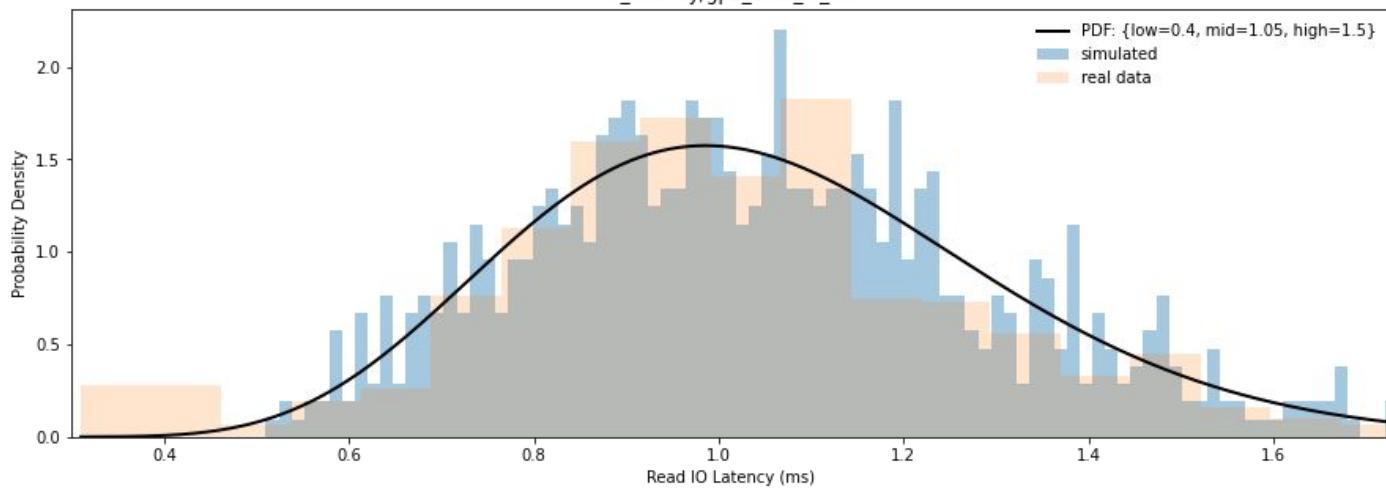
```
$ sudo /usr/share/bcc/tools/biosnoop > ios
$ grep SharedPool ios | tr -s ' ' | cut -f 8 -d ' ' > io_lat
$ cat io_lat | histogram.py -l -p
# NumSamples = 107517; Min = 0.06; Max = 2.43
# Mean = 0.118898; Variance = 0.002304; SD = 0.048005; Median 0.100000
# each █ represents a count of 569
  0.0600 -    0.0623 [    94]:  (0.09%)
  0.0623 -    0.0670 [     0]:  (0.00%)
  0.0670 -    0.0762 [   505]:  (0.47%)
  0.0762 -    0.0948 [ 33459]: ████ ████████████████████ (31.12%)
  0.0948 -    0.1318 [ 42706]: ████ ████████████████████
█ █ (39.72%)
  0.1318 -    0.2060 [ 29154]: ████ ████████████████████ (27.12%)
  0.2060 -    0.3542 [   994]: █ (0.92%)
  0.3542 -    0.6508 [   523]: (0.49%)
  0.6508 -    1.2438 [    77]: (0.07%)
  1.2438 -    2.4300 [     5]: (0.00%)
```

Solution!

io_latency/m5d_cass_io_lat



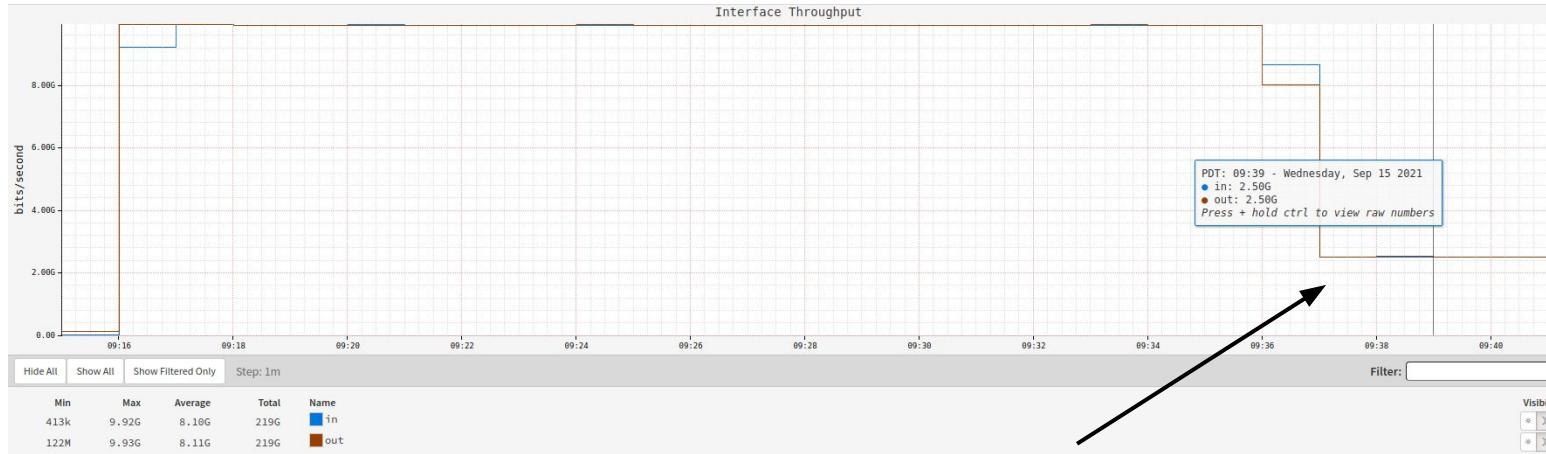
io_latency/gp2_cass_io_lat



Solution!

```
$ iperf3 -s -p 8888
-----
Server listening on 8888
-----
```

```
$ iperf3 -c [REDACTED] -P $(getconf _NPROCESSORS_ONLN) -p 8888 -t 3600
Connecting to host 100.67.65.24, port 8888
[  4] local [REDACTED] port 25344 connected to [REDACTED] port 8888
[  6] local [REDACTED] port 25346 connected to [REDACTED] port 8888
[  8] local [REDACTED] port 25348 connected to [REDACTED] port 8888
[ 10] local [REDACTED] port 25350 connected to [REDACTED] port 8888
[ 12] local [REDACTED] port 25352 connected to [REDACTED] port 8888
[ 14] local [REDACTED] port 25354 connected to [REDACTED] port 8888
[ 16] local [REDACTED] port 25356 connected to [REDACTED] port 8888
[ 18] local [REDACTED] port 25358 connected to [REDACTED] port 8888
```



Record Baseline NOT Burst

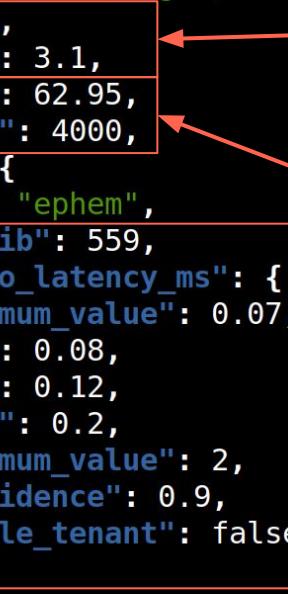
Solution!

```
» cat service_capacity_modeling/hardware/profiles/shapes/aws.json | jq '.instances["m5d.4xlarge"]'  
{  
  "name": "m5d.4xlarge",  
  "cpu": 16,  
  "cpu_ghz": 3.1,  
  "ram_gib": 62.95,  
  "net_mbps": 4000,  
  "drive": {  
    "name": "ephem",  
    "size_gib": 559,  
    "read_io_latency_ms": {  
      "minimum_value": 0.07,  
      "low": 0.08,  
      "mid": 0.12,  
      "high": 0.2,  
      "maximum_value": 2,  
      "confidence": 0.9,  
      "single_tenant": false  
    }  
  }  
}
```

CPU Count and Frequency

Actual Memory and Network

Actual Disk and Disk Latency



https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/hardware/profiles/shapes/aws.json

Solution!

```
» cat service_capacity_modeling/hardware/profiles/pricing/aws/3yr-reserved.json | jq '.["us-east-1"].instances["m5d.4xlarge"]'  
{"annual_cost": 2977.7}  
» cat service_capacity_modeling/hardware/profiles/pricing/aws/3yr-reserved.json | jq '.["us-east-1"].drives'  
[  
  "gp2": {  
    "annual_cost_per_gib": 1.2  
  },  
  "gp3": {  
    "annual_cost_per_gib": 0.96,  
    "annual_cost_per_read_io": 0.005  
  }  
]  
» cat service_capacity_modeling/hardware/profiles/pricing/aws/3yr-reserved.json | jq '.["us-east-1"].services'  
[  
  "blob.standard": {  
    "annual_cost_per_gib": "0.252",  
    "annual_cost_per_write_io": "0.000005",  
    "annual_cost_per_read_io": "0.000004"  
  }  
]
```

Your prices

There are relevant services other than drives

Solution!

```
» cat service_capacity_modeling/hardware/profiles/pricing/aws/3yr-reserved.json | jq '.["us-east-1"].instances["m5d.2xlarge"]'  
{  
  "annual_cost": 1488.6,  
  "lifecycle": "stable"  
}  
» cat service_capacity_modeling/hardware/profiles/pricing/aws/3yr-reserved.json | jq '.["us-east-1"].instances["r5n.2xlarge"]'  
{  
  "annual_cost": 1840.3,  
  "lifecycle": "alpha"  
}  
» cat service_capacity_modeling/hardware/profiles/pricing/aws/3yr-reserved.json | jq '.["us-east-1"].instances["i3.2xlarge"]'  
{  
  "annual_cost": 2312,  
  "lifecycle": "deprecated"  
}
```

Company specific lifecycle

Capacity Planning 201

$$M(D, H, PL) \rightarrow C$$

M = Workload Capacity Model

D = User Desire

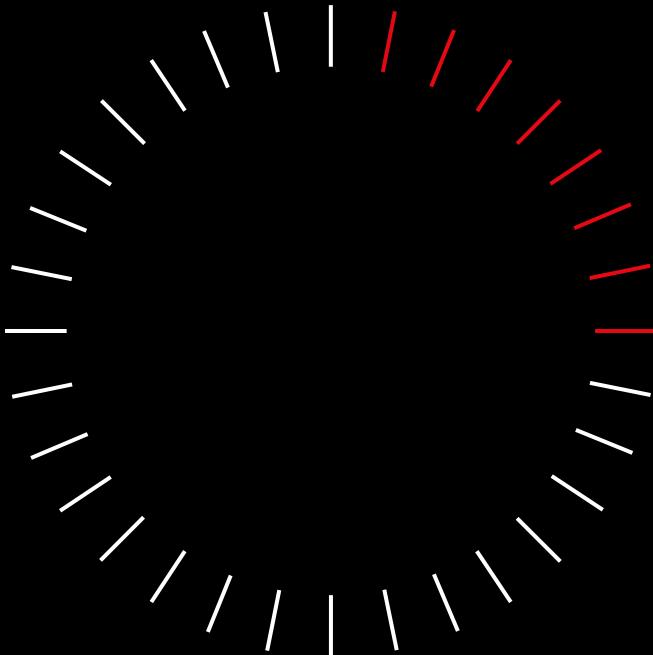
H = Hardware Profile

PL = Current Pricing and Lifecycle

C = Candidate Cluster

User Input

We need a unified language for talking about requirements



User Input

The user probably knows how much CPU, RAM, Network, Disk space, and Disk IOs they need

User Input

The user probably knows how much CPU, RAM, Network, Disk space, and Disk IOs they need

They probably don't

User Input

The user probably knows how much CPU, RAM, Network, Disk space, and Disk IOs they need

They probably don't

Well they must know how much traffic they will send, how big their data is?

User Input

The user probably knows how much CPU, RAM, Network, Disk space, and Disk IOs they need

They probably don't

Well they must know how much traffic they will send, how big their data is?

They probably don't

User Input

We will never know the truth

User Input

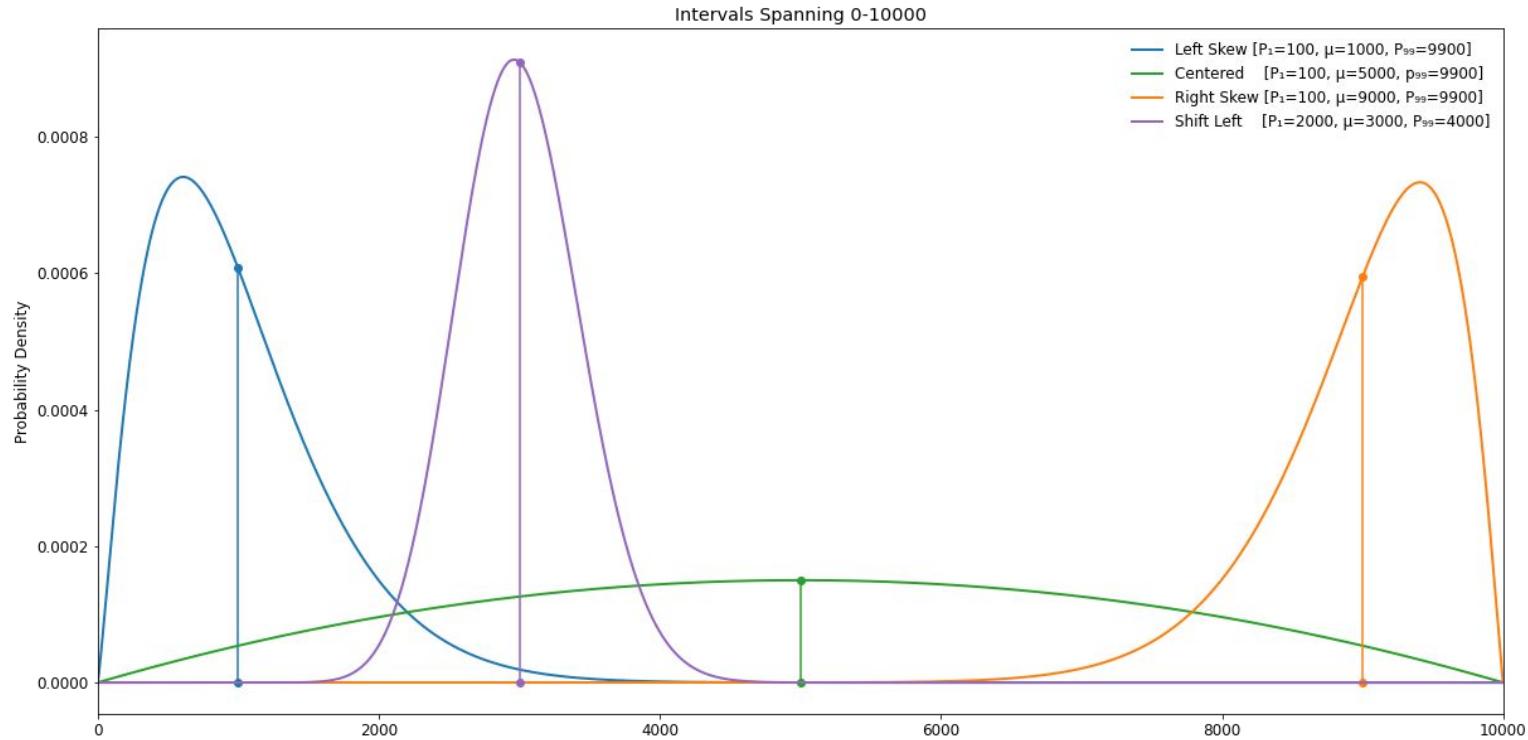
The user probably knows how much CPU, RAM, Network, Disk space, and Disk IOs they need

They probably don't

Well they must know how much traffic they will send, how big their data is?

They probably *don't know exactly*

Intervals



```
left_skew = Interval(minimum_value=0, low=100, mid=1000, high=9900, maximum_value=10000, confidence=0.98)
right_skew = Interval(minimum_value=0, low=100, mid=9000, high=9900, maximum_value=10000, confidence=0.98)
center = Interval(minimum_value=0, low=100, mid=5000, high=9900, maximum_value=10000, confidence=0.98)
shift = Interval(minimum_value=0, low=2000, mid=3000, high=4000, maximum_value=10000, confidence=0.98)
```

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/stats.py#L90-L138

Capacity Desires

```
# How critical is this cluster, impacts how much "extra" we provision
# 0 = Critical to the product          (Product does not function)
# 1 = Important to product with fallback (User experience degraded)
# 2 = Care about it but don't wake up    (Internal apps)
# 3 = Do not care                      (Testing)
service_tier: int = 1

# How will the service be queried
query_pattern: QueryPattern = QueryPattern()

# What will the state look like
data_shape: DataShape = DataShape()

# When users are providing latency estimates, what is the typical
# instance core frequency we are comparing to. Databases use i3s a lot
# hence this default
core_reference_ghz: float = 2.3
```

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/interface.py#L448-L465

Capacity Desires

```
# How critical is this cluster, impacts how much "extra" we provision
# 0 = Critical to the product          (Product does not function)
# 1 = Important to product with fallback (User experience degraded)
# 2 = Care about it but don't wake up    (Internal apps)
# 3 = Do not care                      (Testing)
service_tier: int = 1

# How will the service be queried
query_pattern: QueryPattern = QueryPattern()

# What will the state look like
data_shape: DataShape = DataShape()

# When users are providing latency estimates, what is the typical
# instance core frequency we are comparing to. Databases use i3s a lot
# hence this default
core_reference_ghz: float = 2.3
```

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/interface.py#L448-L465

How sad are you if this cluster fails?

Tier 0 =  ≪ 

Tier 1 =  ≪ 

Tier 2 =  ≈ 

Tier 3 =  ≫ 

Query Pattern

```
# Will the service primarily be accessed in a latency sensitive mode
# (aka we care about P99) or throughput (we care about averages)
access_pattern: AccessPattern = AccessPattern.latency
access_consistency: GlobalConsistency = GlobalConsistency()

# A main input, how many requests per second will we handle
# We assume this is the mean of a range of possible outcomes
estimated_read_per_second: Interval = certain_int(0)
estimated_write_per_second: Interval = certain_int(0)

# A main input, how much _on cpu_ time per operation do you take.
# This depends heavily on workload, but this is a generally ok default
# For a Java app (C or C++ will generally be about 10x better,
# python 2-4x slower, etc...)
estimated_mean_read_latency_ms: Interval = certain_float(1)
estimated_mean_write_latency_ms: Interval = certain_float(1)

# For stateful services the amount of data accessed per
# read and write impacts disk and network provisioning
# For stateless services it mostly just impacts memory and network
estimated_mean_read_size_bytes: Interval = certain_int(AVG_ITEM_SIZE_BYTES)
estimated_mean_write_size_bytes: Interval = certain_int(AVG_ITEM_SIZE_BYTES /

# The latencies at which oncall engineers get involved. We want
# to provision such that we don't involve oncall
# Note that these summary statistics will be used to create reasonable
# distribution approximations of these operations (yielding p25, p99, etc)
read_latency_slo_ms: FixedInterval = FixedInterval(
    low=0.4, mid=4, high=10, confidence=0.98
)
write_latency_slo_ms: FixedInterval = FixedInterval(
    low=0.4, mid=4, high=10, confidence=0.98
)
```

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/service_capacity_modeling/interface.py#L372-L405

How is it queried?

read/write
sizing
latency

Provide defaults from
the model

Inputs are Intervals

Data Shape

```
estimated_state_size_gib: Interval = certain_int(0)
estimated_state_item_count: Optional[Interval] = None
estimated_working_set_percent: Optional[Interval] = None

# How compressible is this dataset. Note that databases might offer
# better or worse compression strategies that will impact this
# Note that the ratio here is the forward ratio, e.g.
# A ratio of 2 means 2:1 compression (0.5 on disk size)
# A ratio of 5 means 5:1 compression (0.2 on disk size)
estimated_compression_ratio: Interval = certain_float(1)

# How much fixed memory must be provisioned per instance for the
# application (e.g. for process heap memory)
reserved_instance_app_mem_gib: int = 2

# How much fixed memory must be provisioned per instance for the
# system (e.g. for kernel and other system processes)
reserved_instance_system_mem_gib: int = 1

# How durable does this dataset need to be. We want to provision
# sufficient replication and backups of data to achieve the target
# durability SLO so we don't lose our customer's data. Note that
# This is measured in orders of magnitude. So
# 1000 = 1 - (1/1000) = 0.999
# 10000 = 1 - (1/10000) = 0.9999
durability_slo_order: FixedInterval = FixedInterval(
    low=1000, mid=10000, high=100000, confidence=0.98
)
```

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/service_capacity_modeling/interface.py#L408-L445

How is the data
shaped?
footprint
durability

Provide defaults from
the model

Inputs are Intervals

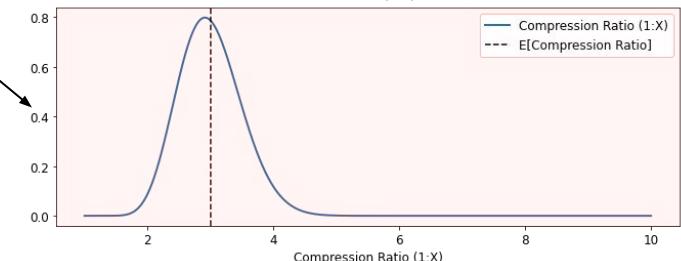
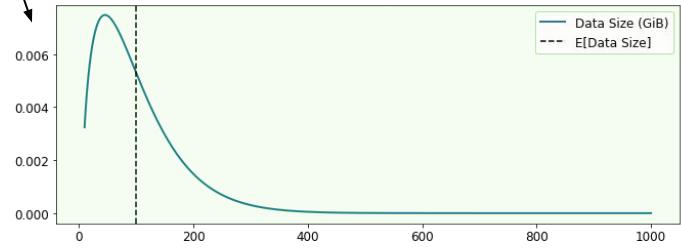
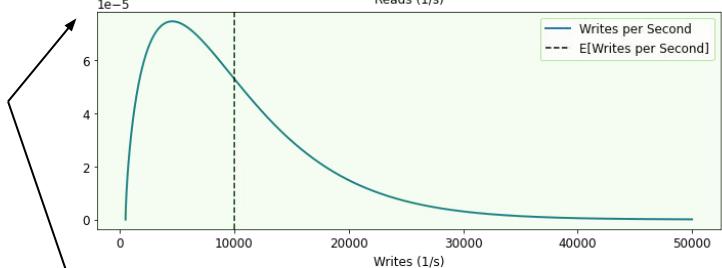
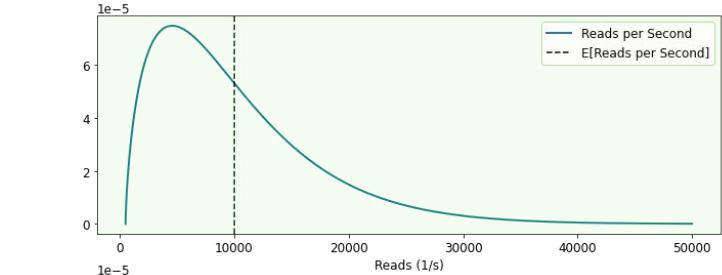
Intervals

```
from service_capacity_modeling.interface import CapacityDesires
from service_capacity_modeling.interface import FixedInterval, Interval
from service_capacity_modeling.interface import QueryPattern, DataShape

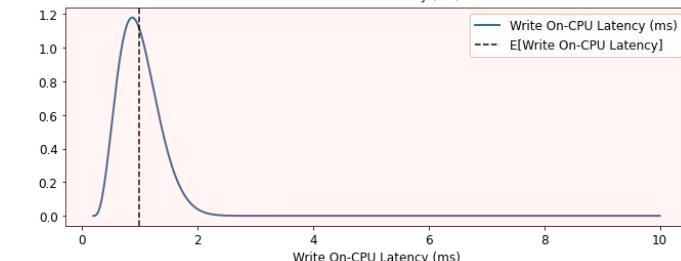
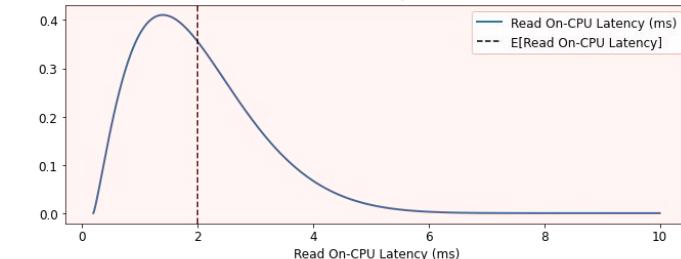
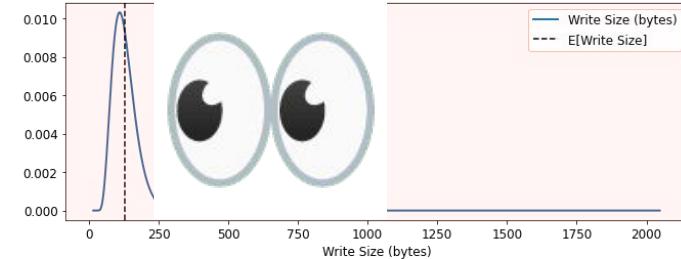
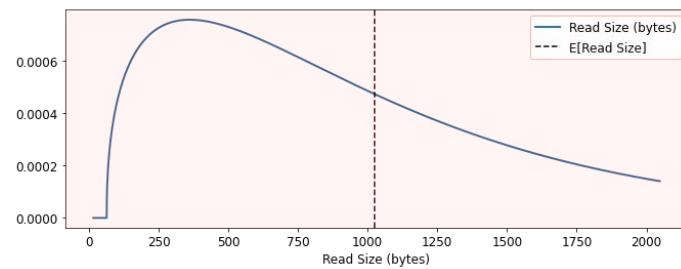
desires = CapacityDesires(
    # This service is critical to the business
    service_tier=1,
    query_pattern=QueryPattern(
        # Not sure exactly how much QPS we will do, but we think around
        # 10,000 reads and 10,000 writes per second.
        estimated_read_per_second=Interval(
            low=1_000, mid=10_000, high=100_000, confidence=0.98
        ),
        estimated_write_per_second=Interval(
            low=1_000, mid=10_000, high=100_000, confidence=0.98
        ),
    ),
    # Not sure how much data, but we think it'll be around 100 GiB
    data_shape=DataShape(
        estimated_state_size_gib=Interval(low=10, mid=100, high=1_000, confidence=0.98),
    ),
)
```

Intervals

From the Human

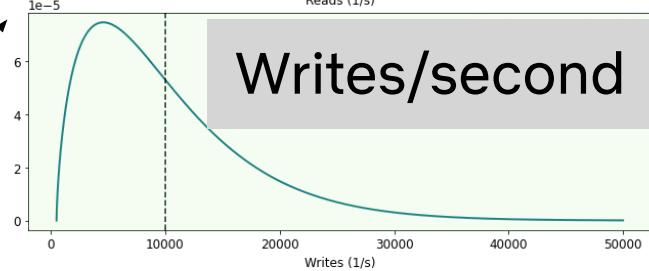
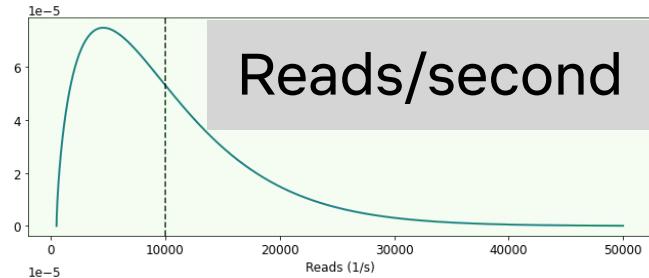


From the Model

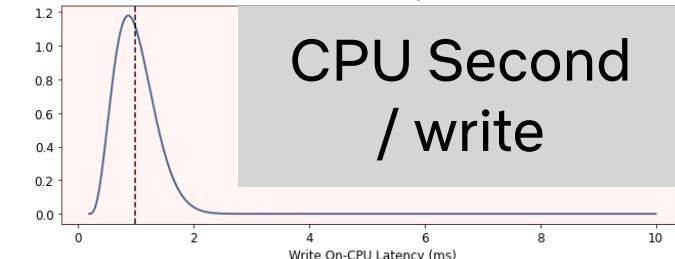
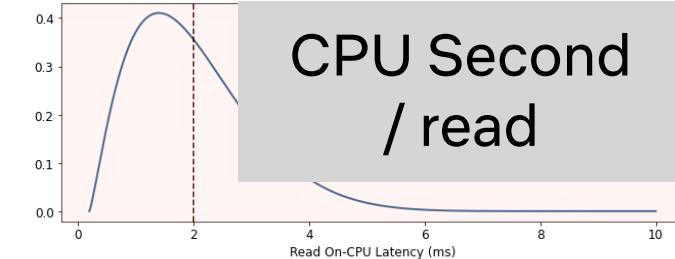
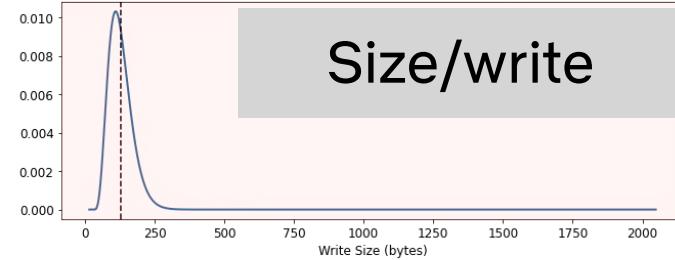
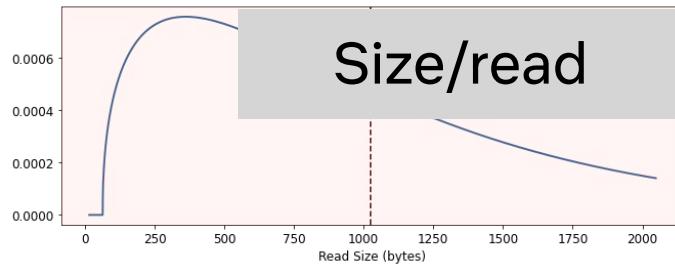
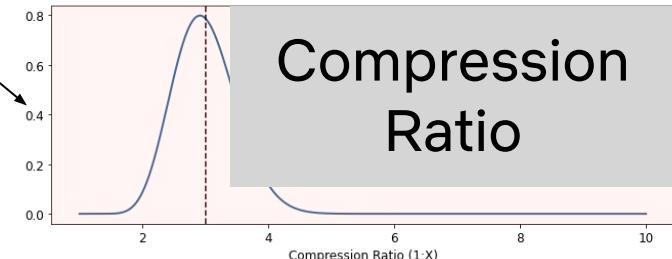
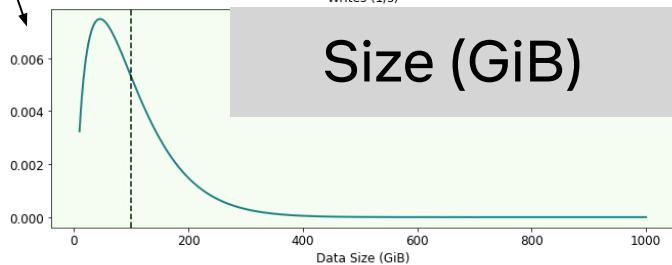


Intervals

From the Human



From the Model



Capacity Planning 301

$$M(D, H, PL) \rightarrow C$$

M = Workload Capacity Model

D = User Desire

H = Hardware Profile

PL = Current Pricing and Lifecycle

C = Candidate Cluster

Capacity Planning Cassandra

Uncertain requirements

Computers cost money

...

Which computers should I buy
For Cassandra?

To do it right we need the
right inputs

And some math ...

Let's do the certain case first

Aka "let's ignore the distributions for a second"

Building a Model

We need to compute a **Cluster** from a **Desire** and **Hardware** context

```
class NflxCassandraCapacityModel(CapacityModel):
    @staticmethod
    def capacity_plan(
        instance: Instance,
        drive: Drive,
        context: RegionContext,
        desires: CapacityDesires,
        extra_model_arguments: Dict[str, Any],
    ) -> Optional[CapacityPlan]:
```

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/models/org/netflix/cassandra.py#L386-L392

CPU

let μ = average $\frac{\text{CPU time}}{\text{request}}$

let λ = average $\frac{\text{request}}{\text{second}}$

$$R = \lambda \times \mu$$

$$\text{CPUs} = R + Q * \sqrt{R}$$

Service Tier	P(Queue)	Q
0	1%	2.375
1	5%	1.761
2	20%	1.16
3	30%	1

15.3 Square-Root Staffing

In this section, we refine the $R + \sqrt{R}$ approximation developed in the previous section.

As before, we assume an M/M/k with average arrival rate λ and average server speed μ . The QoS goal that we set is that P_Q , the probability of queueing in the M/M/k, should be below some given value α (e.g., $\alpha = 20\%$). Our goal is to determine the minimal number of servers, k_α^* , needed to meet this QoS goal.

Note that bounding P_Q is really equivalent to bounding mean response time or mean queueing time, or similar metrics, because they are all simple functions of P_Q (e.g., from (14.9), we have $\mathbf{E}[T_Q] = \frac{1}{\lambda} \cdot P_Q \cdot \frac{\rho}{1-\rho}$).

Theorem 15.2 (Square-Root Staffing Rule) *Given an M/M/k with arrival rate λ and server speed μ and $R = \lambda/\mu$, where R is large, let k_α^* denote the least number of servers needed to ensure that $P_Q^{\text{M/M/k}} < \alpha$. Then*

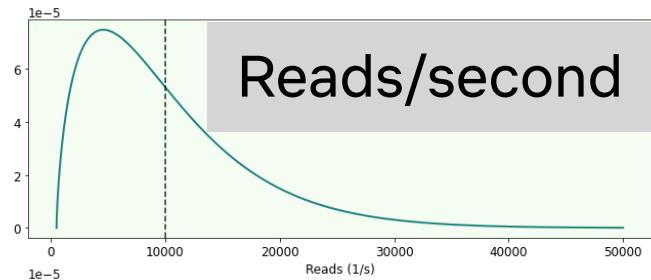
$$k_\alpha^* \approx R + c\sqrt{R},$$

where c is the solution to the equation,

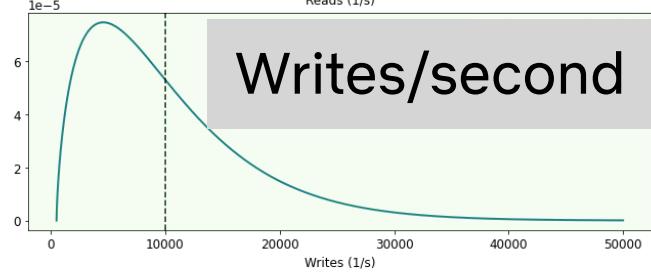
$$\frac{c\Phi(c)}{\phi(c)} = \frac{1 - \alpha}{\alpha} \tag{15.4}$$

where $\Phi(\cdot)$ denotes the c.d.f. of the standard Normal and $\phi(\cdot)$ denotes its p.d.f.

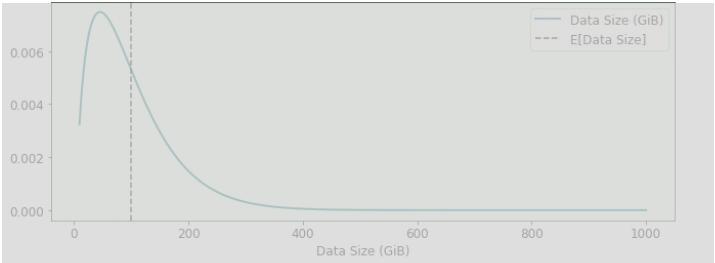
CPU



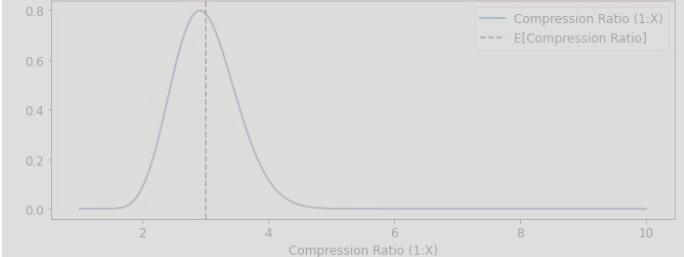
Reads/second



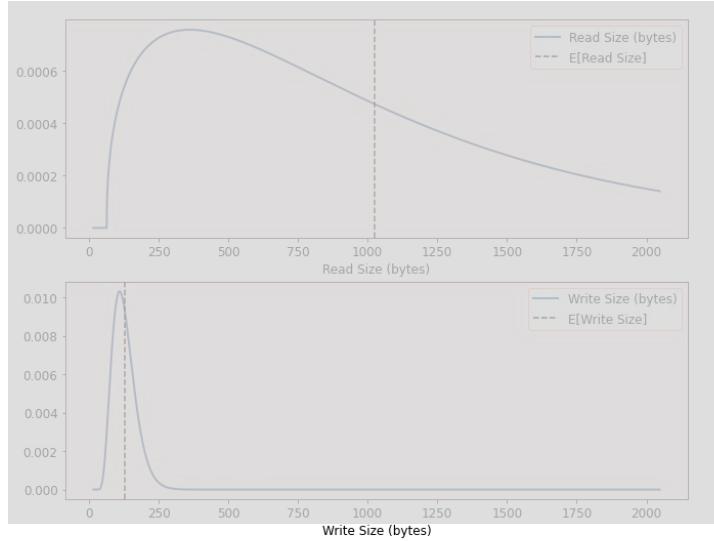
Writes/second



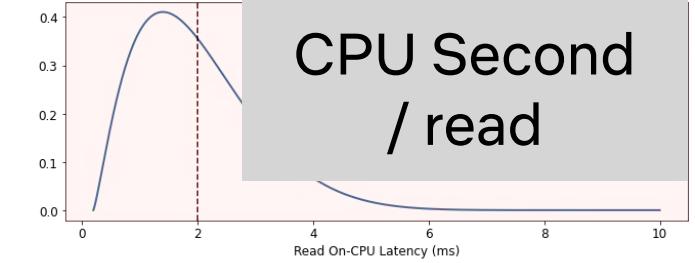
Data Size (GiB)
--- E[Data Size]



Compression Ratio (1:X)
--- E[Compression Ratio]

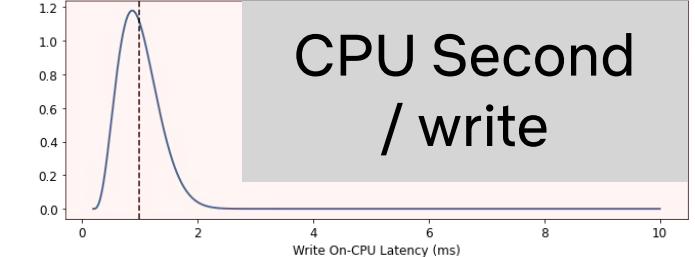


Read Size (bytes)
--- E[Read Size]



Write Size (bytes)
--- E[Write Size]

CPU Second
/ read



CPU Second
/ write

Network

For simple case it's easy

Tricky in complex case...

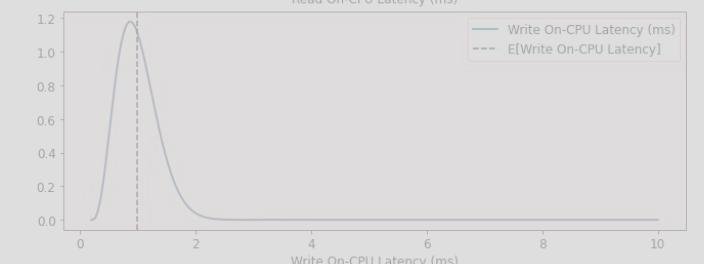
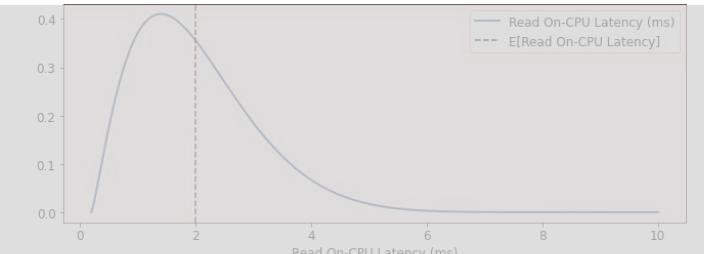
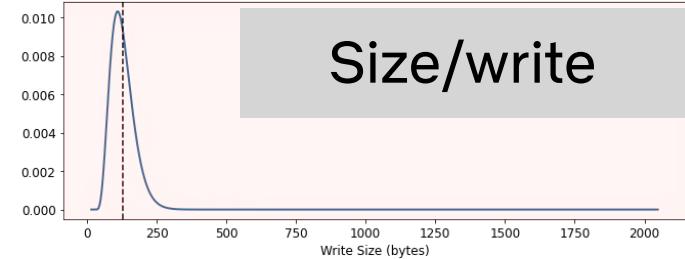
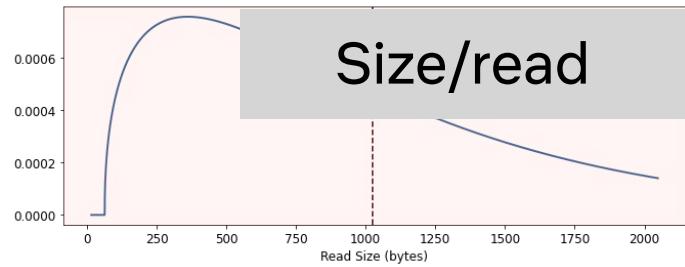
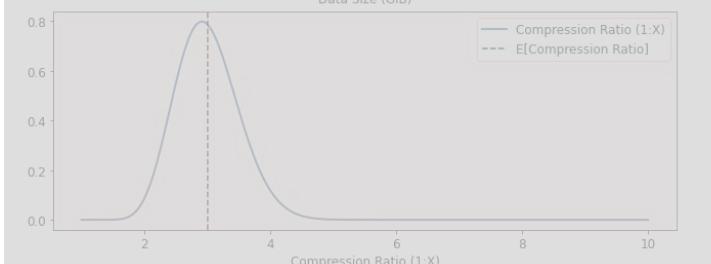
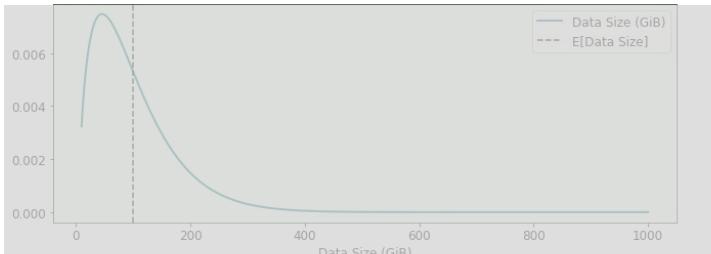
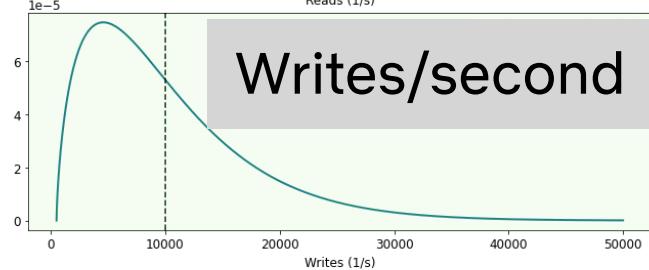
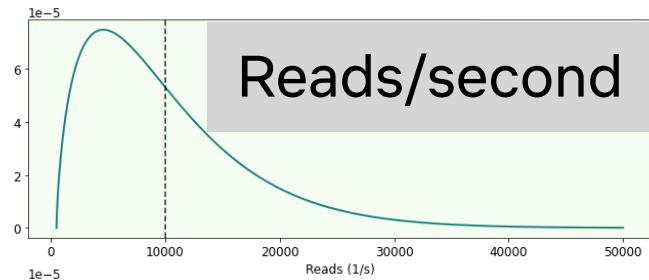
We have to know Consistency Level and Replication Factor

$$\text{let } \mu_r = \frac{\text{bytes}}{\text{read}}$$
$$\text{let } \lambda_r = \frac{\text{read}}{\text{second}}$$
$$\text{let } \mu_w = \frac{\text{bytes}}{\text{write}}$$
$$\text{let } \lambda_w = \frac{\text{write}}{\text{second}}$$

$$BW_{\text{simple}} = K \times (\mu_r \times \lambda_r + \mu_w \times \lambda_w)$$

$$BW_{\text{complex}} = K \times (CL \times (\mu_r \times \lambda_r) + RF \times (\mu_w \times \lambda_w))$$

Network



Disk

Compaction strategy
and Compression
matter

Tricky: Remember
network drives must be
sized for IO

$$\text{size}_{\text{zone}} = \frac{\text{RF} \times \text{data size}}{\#\text{zones} \times \text{compression}}$$

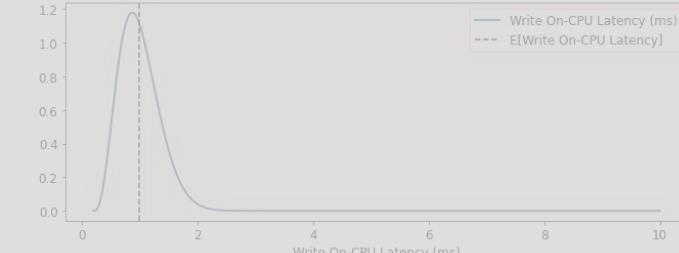
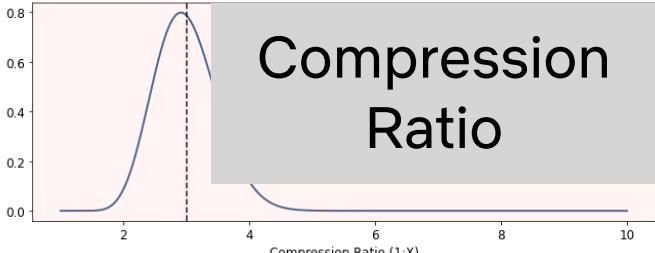
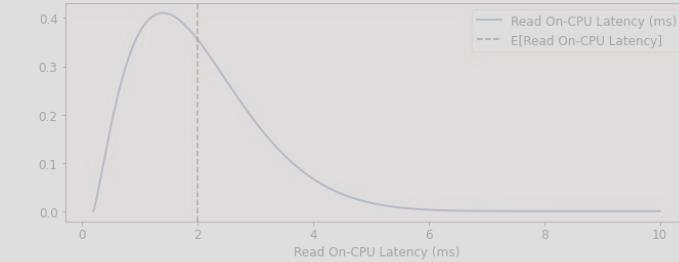
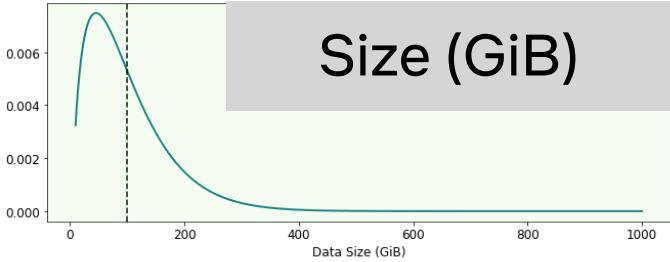
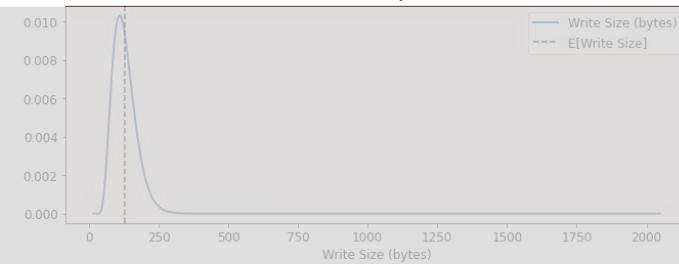
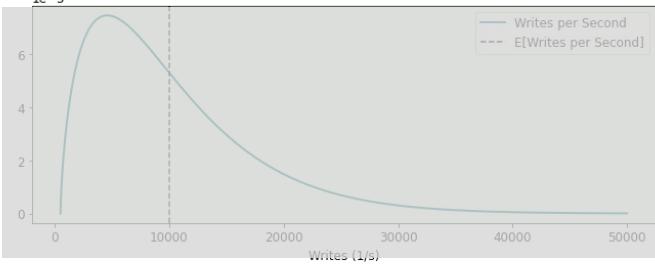
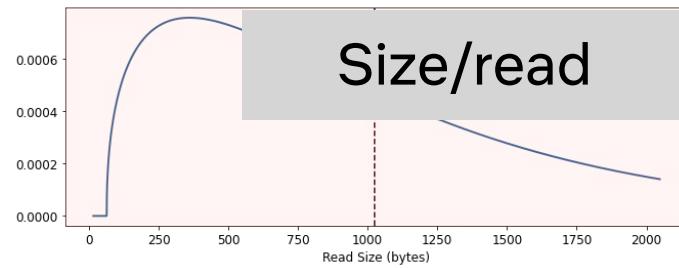
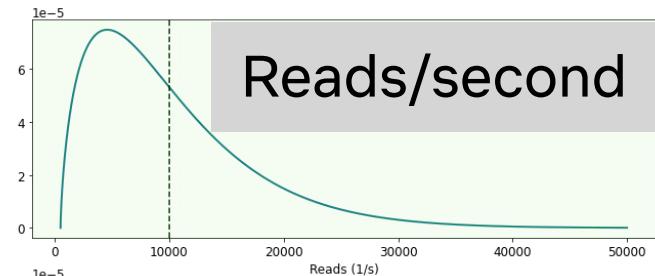
$$\text{size}_{\text{node}} = \frac{\text{size}_{\text{zone}}}{\#\text{nodes}_{\text{zone}}} \times f(\text{compaction})$$

$$\text{Epem}_{\text{node}} = \text{size}_{\text{node}}$$

OR

$$\text{EBS}_{\text{node}} = \max(\text{size}_{\text{node}}, f(\text{read BW}))$$

Disk



Memory

Fundamental Tradeoff
reads (page cache) or
writes (heap)

Tricky: This depends
on the number of
nodes.

$$RAM_{\text{read}} = f(\text{data size, working set})$$

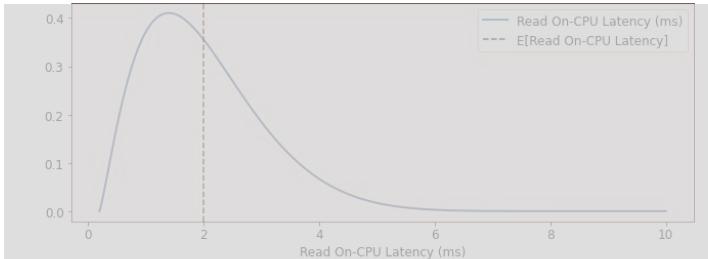
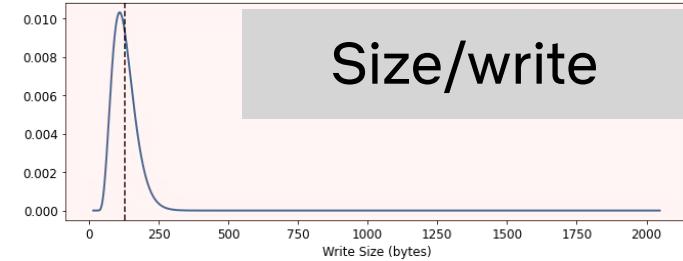
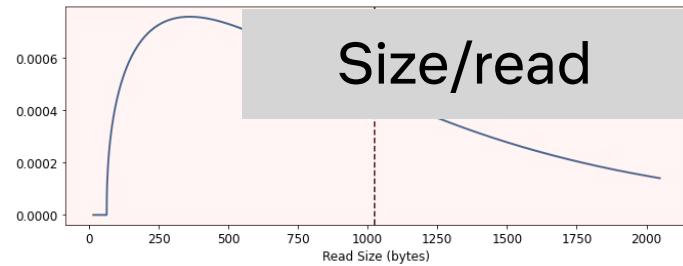
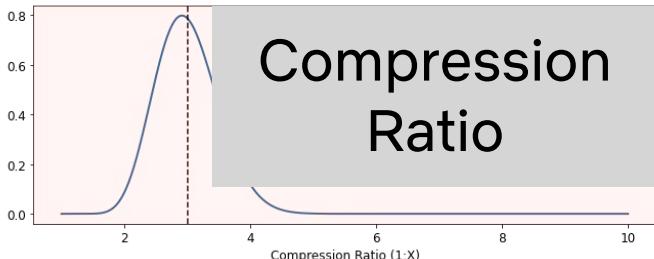
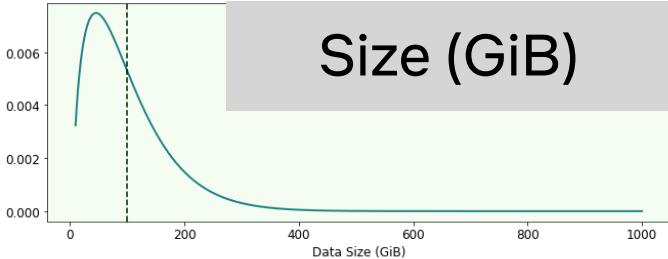
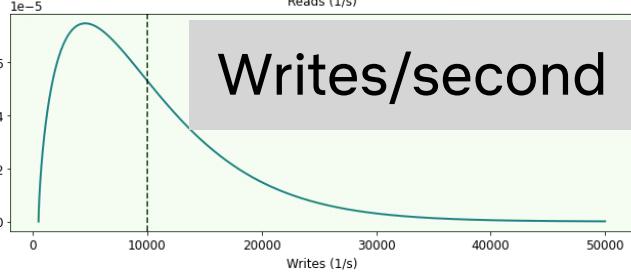
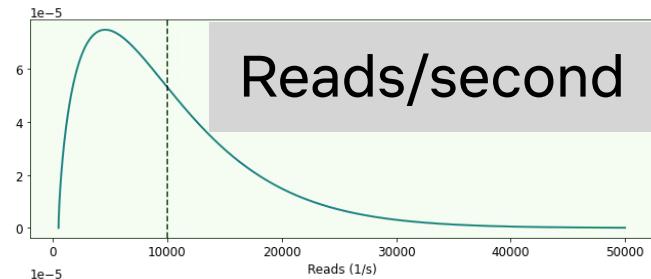
$$RAM_{\text{write}} = f(\text{write BW, compaction})$$

$$RAM_{\text{JVM}} = f(\text{write BW, read BW})$$

$$RAM_{\text{system}} = f(\text{sidecars, kernel})$$

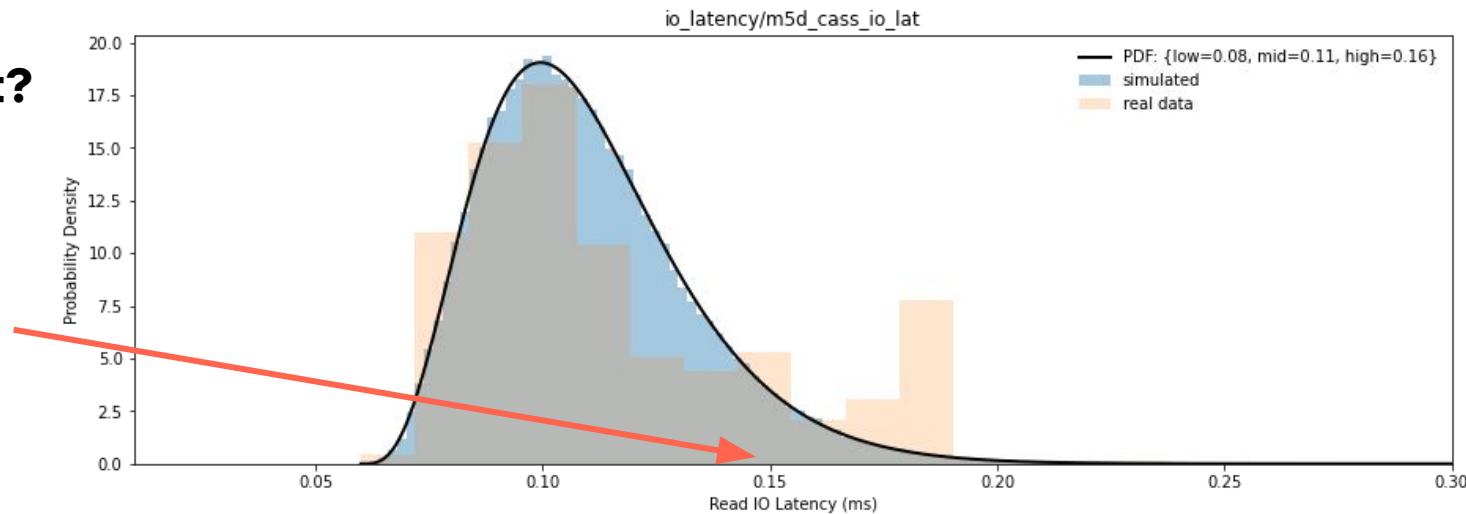
$$RAM = \sum RAM_{\text{component}}$$

Memory

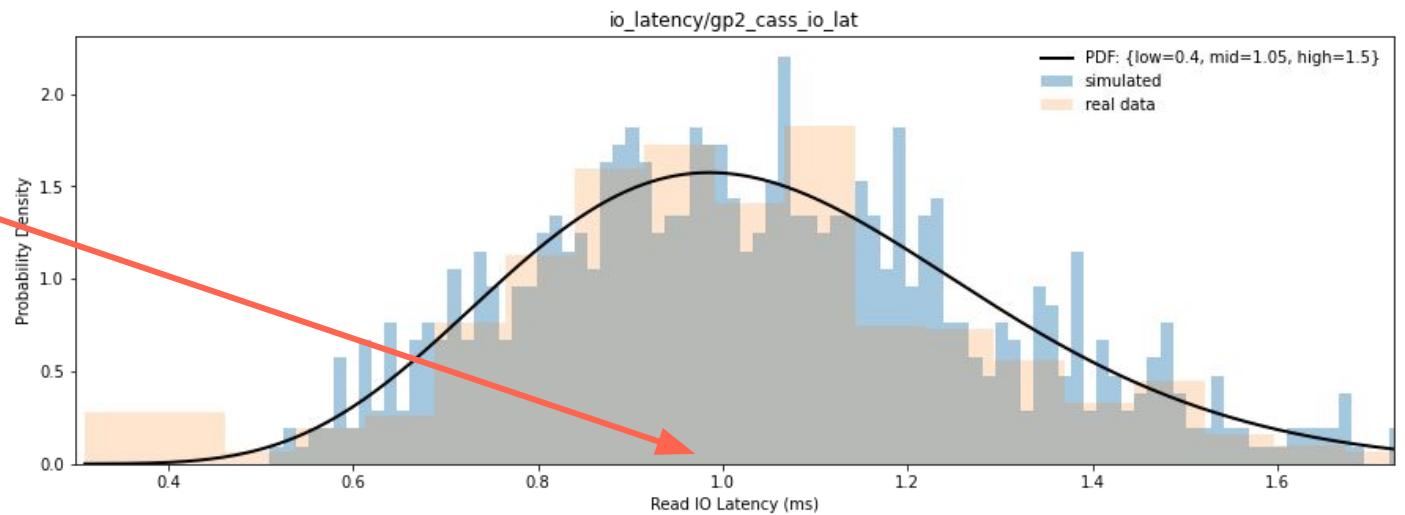


Working Set?

This needs
very little
RAM



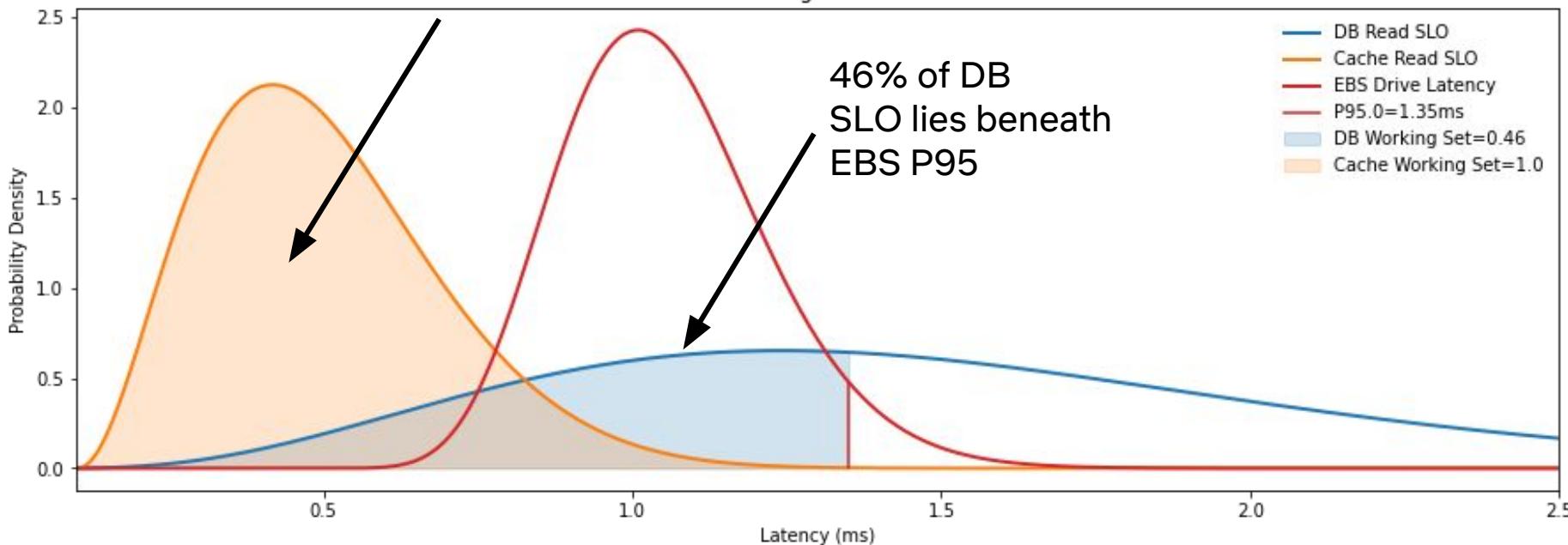
This needs
more
RAM



Working Set

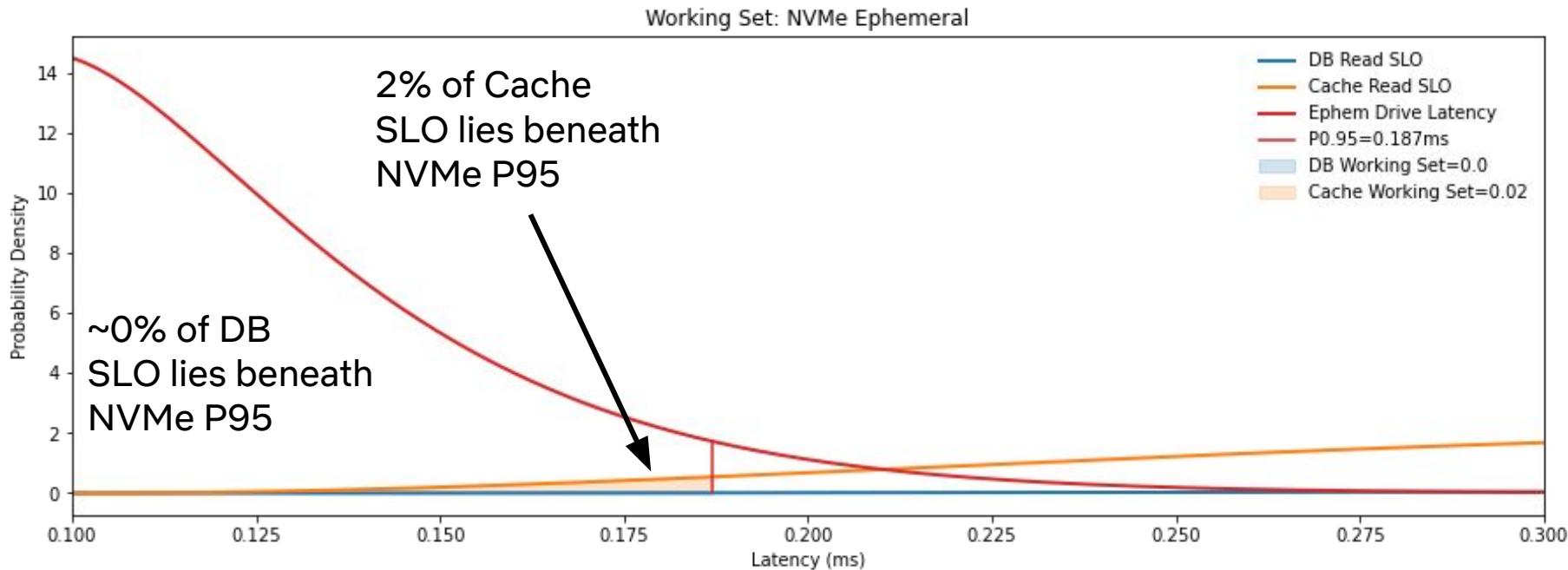
>99% of Cache
SLO lies beneath
EBS P95

Working Set: EBS



https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/models/common.py#L238-L276

Working Set



https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/models/common.py#L238-L276

Success!

We can compute a cluster
for a given input.

Success!

We can compute a cluster
for a given input.

But we have dozens of
hardware types and cloud
drives ...

Capacity Planning 301

$$\forall H(M_{cassandra}(D, H, P)) \rightarrow C_H$$

choose

$$C = \operatorname{argmin}_H(cost(C_H))$$

$M(D, m5.2xlarge) \rightarrow 12 \text{ m5.2xlarge} + 200GiB gp2$

$M(D, m5.4xlarge) \rightarrow 6 \text{ m5.4xlarge} + 400GiB gp2$

$M(D, r5d.2xlarge) \rightarrow 6 \text{ r5d.2xlarge}$

... now pick the cheapest one

Great Success!

We can compute a cluster over all inputs.

Great Success!

We can compute a cluster over all inputs.

But our inputs are *distributions* ... and we have like 20 of them ...

Capacity Planning

Take it to 11

Time for some Monte Carlo



Let's **simulate** possible **worlds**

Take it to 11

Get some tail events

And pick the **choice of least
regret** across all worlds

What do we regret?

Money for hardware

- Bought too little
- Bought too much

Running out of Disk

And ... more (pluggable)

$$\text{regret}(X, Y)_{\$} = K_{\$}(X_{\$} - Y_{\$})^{r_{\$}}$$

$$\text{regret}(X, Y)_{\text{disk}} = K_{\text{disk}}(X_{\text{disk}} - Y_{\text{disk}})^{r_{\text{disk}}}$$

$$\text{regret}(X, Y) = \sum_i \text{regret}(X, Y)_i$$

$$\text{regret}(X_i) = \sum_j^X \text{regret}(X_i, X_j)$$

$$\text{regret}_{\text{least}} = \operatorname{argmin}_X (\text{regret})$$

World 1

We buy
48 i3en.xlarge costing
\$73,652.57

We require 6,634.0 GiB

World 2

We buy
96 r5.8xlarge costing
\$646,309.93

We require 17,941 GiB

World 1

IN

World 2

We **bought**
48 i3en.xlarge costing
\$73,652.57

We **have** 6,634.0 GiB

We **needed to buy**
96 r5.8xlarge costing
\$646,309.93

We **required** 17,941 GiB

$$\text{regret}(W_1 \text{ in } W_2)_{\$} = 1.25 \times |73,652.57 - 646,309.93|^{1.2} \approx 10M$$

$$\text{regret}(W_1 \text{ in } W_2)_{\text{disk}} = 1.10 \times |6,634.0 - 17,941|^{1.05} \approx 20K$$

regret(W_1 in W_2) \approx 10 million dollars (underprovisioned)

World 2

IN

World 1

We **bought**
96 r5.8xlarge costing
\$646,309.93

We **needed to buy**
48 i3en.xlarge costing
\$73,652.57

We **have** 17,941 GiB

We **required** 6,634.0 GiB

$$\text{regret}(W_2 \text{ in } W_1)_{\$} = 1.0 \times |73,652.57 - 646,309.93|^{1.2} \approx 8M$$

$$\text{regret}(W_2 \text{ in } W_1)_{\text{disk}} = 0.0 \times |6,634.0 - 17,941|^{1.05} = 0K$$

regret(W_2 in W_1) \approx 8 million dollars (overprovisioned)

**Regret is not
symmetric!**

Choice of constants
determines relative cost of

Under-provisioning
(buying too little)

versus **over-provisioning**
(buying too much)

Least Regret

```
desires = CapacityDesires(  
    # This service is critical to the business  
    service_tier=1,  
    query_pattern=QueryPattern(  
        # Not sure exactly how much QPS we will do, but we think around  
        # 10,000 reads and 10,000 writes per second.  
        estimated_read_per_second=Interval(  
            low=1_000, mid=10_000, high=100_000, confidence=0.98  
        ),  
        estimated_write_per_second=Interval(  
            low=1_000, mid=10_000, high=100_000, confidence=0.98  
        ),  
    ),  
    # Not sure how much data, but we think it'll be around 100 GiB  
    data_shape=DataShape(  
        estimated_state_size_gib=Interval(  
            low=10, mid=100, high=1_000, confidence=0.98  
        ),  
    ),  
)
```

Least Regret

```
from service_capacity_modeling.capacity_planner import planner
from service_capacity_modeling.models.org import netflix

# Load up the Netflix capacity models
planner.register_group(netflix.models)

# Plan a cluster
plan = planner.plan(
    model_name="org.netflix.cassandra",
    region="us-east-1",
    desires=desires,
    simulations=1024,
    explain=True
)
```

Least Regret Choice:

12 m5d.xlarge costing 8973.94

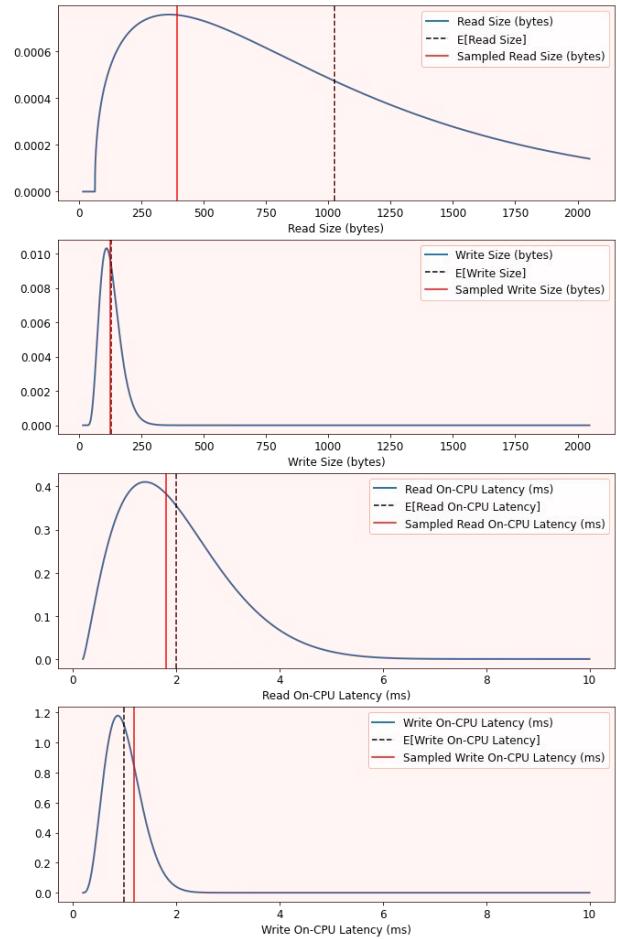
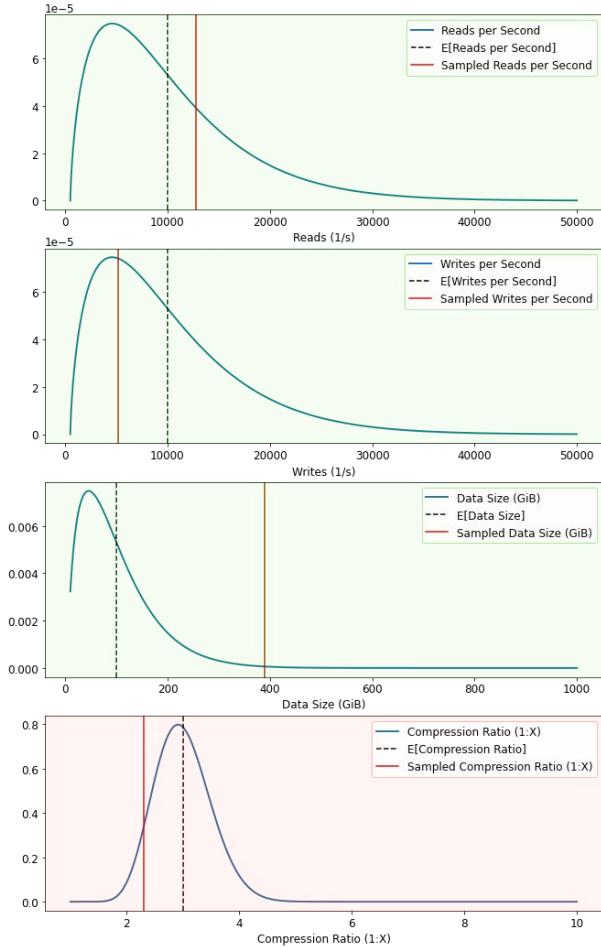
All Choices

{' 6 r5.xlarge': 4,
' 6 r5d.large': 31,
' 6 m5.2xlarge': 2,
' 6 m5d.xlarge': 224,
' 12 m5.xlarge': 132,
' 12 m5d.xlarge': 277,
' 24 m5.xlarge': 242,
' 24 m5d.xlarge': 54,
' 48 m5.xlarge': 55,
' 48 m5d.xlarge': 2,
' 96 m5.xlarge': 1}

Least Regret World

12 m5d.xlarge

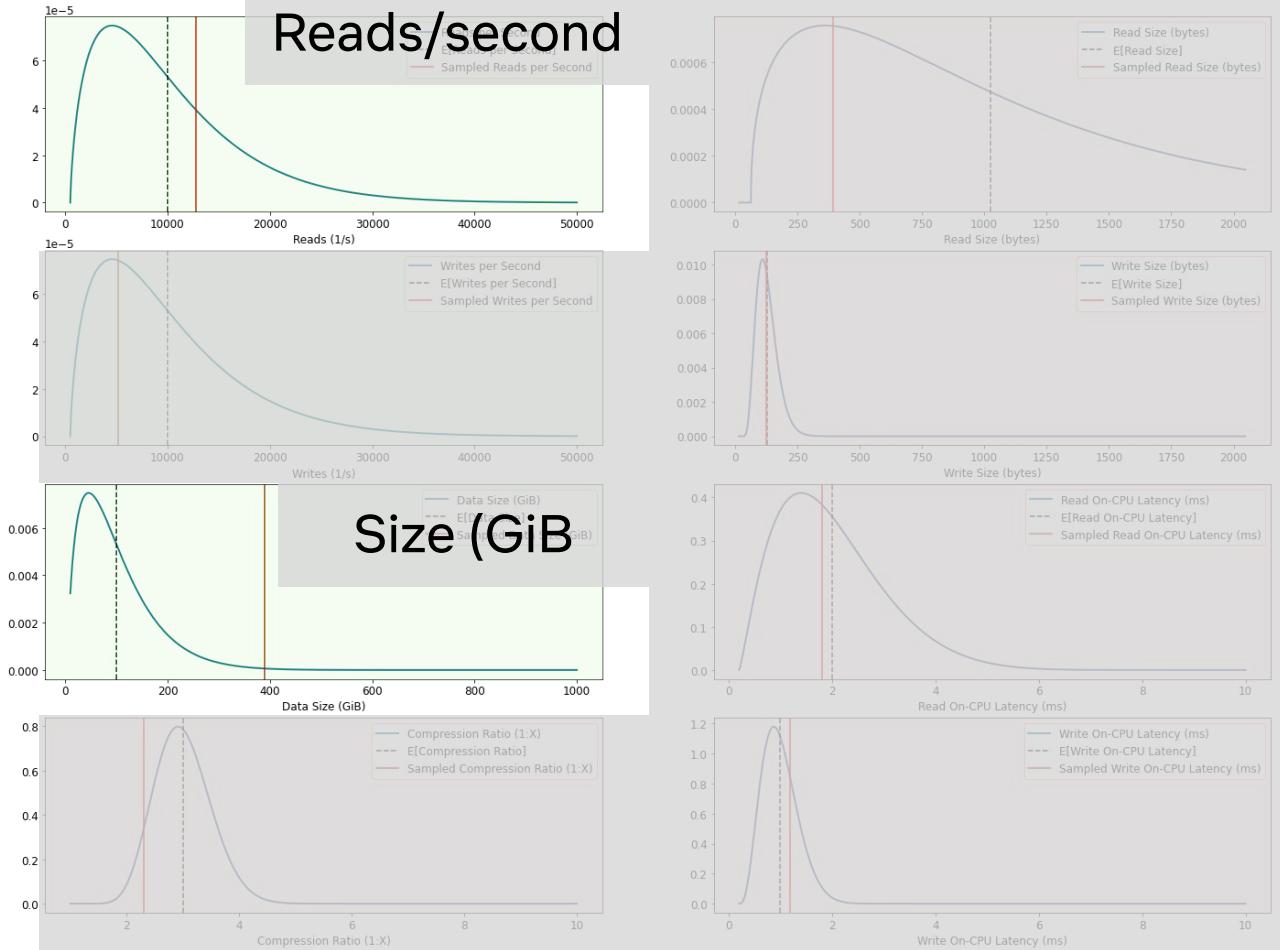
\$8,973.94
per year



Least Regret World

12 m5d.xlarge

\$8,973.94
per year

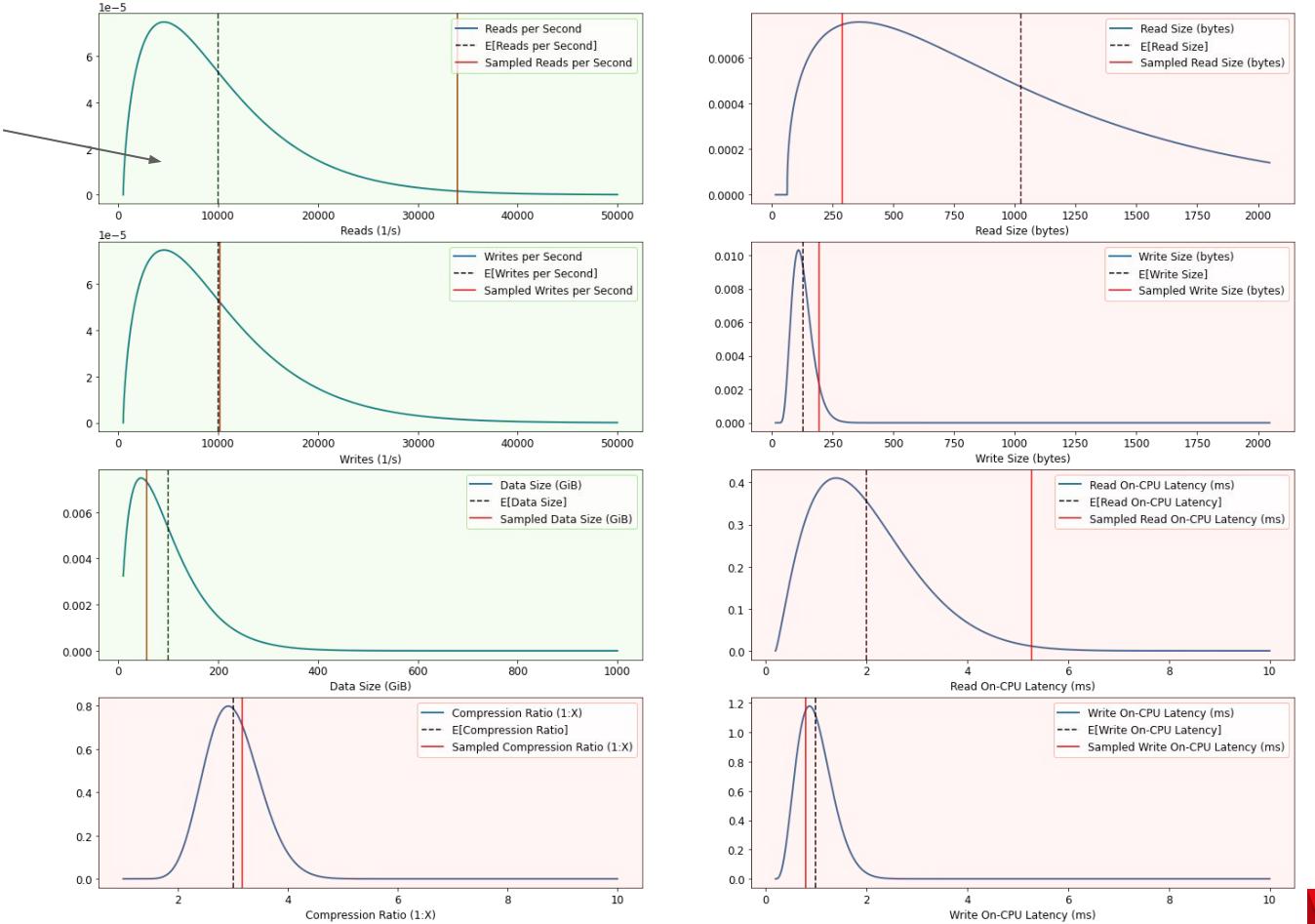


Highest Regret World

**96 m5.xlarge
with 400 GiB gp2**

\$62,145.34

Overprovisioned!

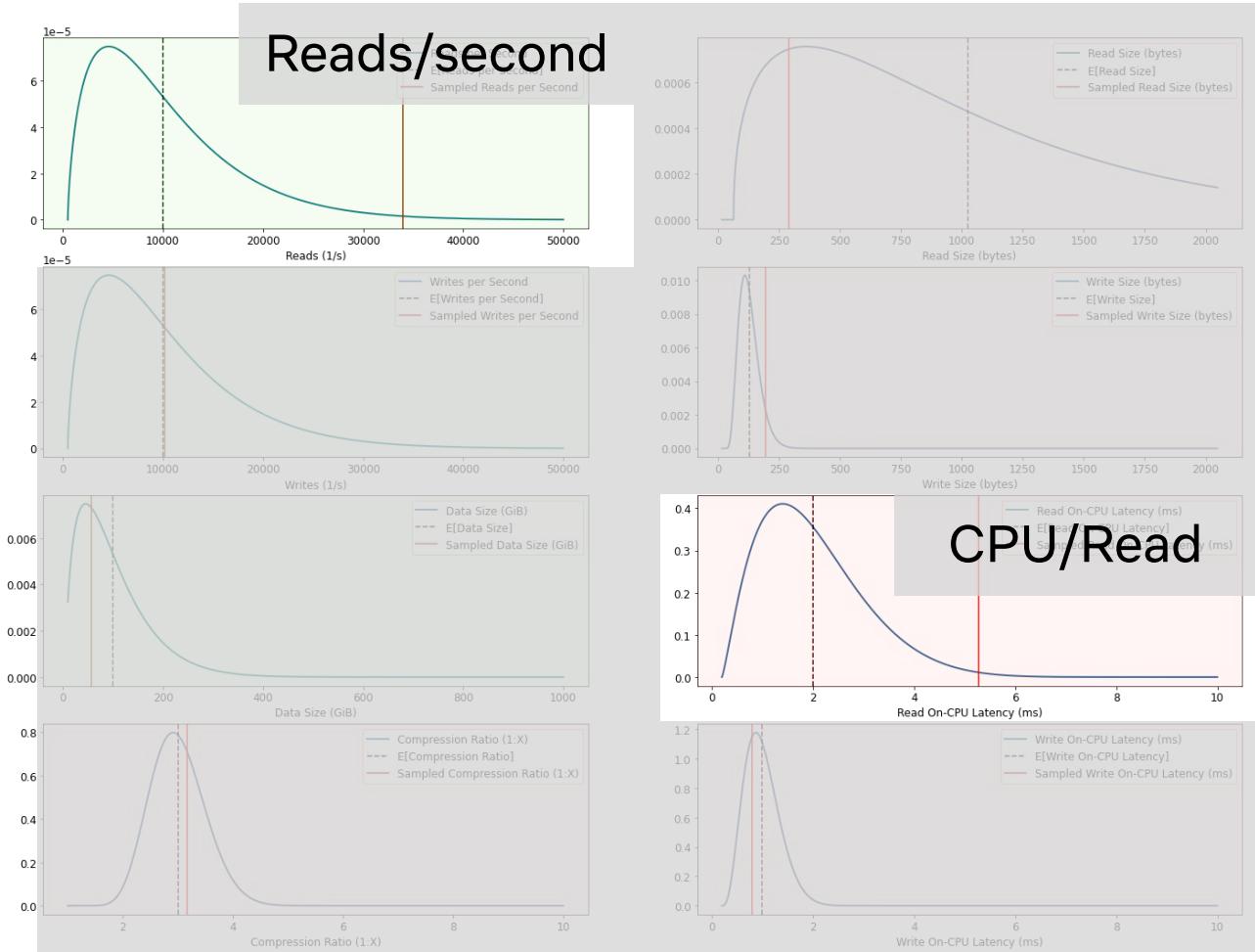


Highest Regret World

96 m5.xlarge
with 400 GiB gp2

\$62,145.34

Overprovisioned!

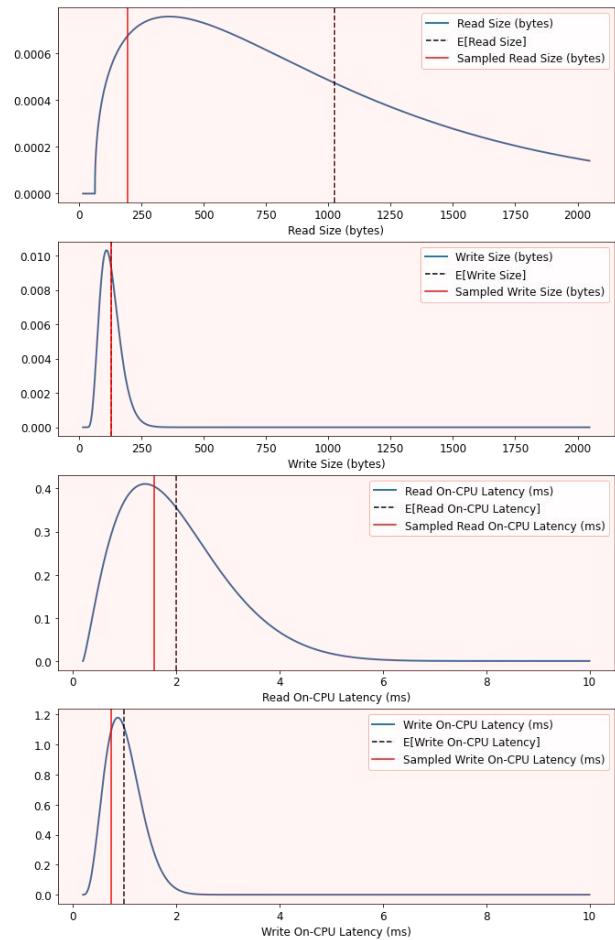
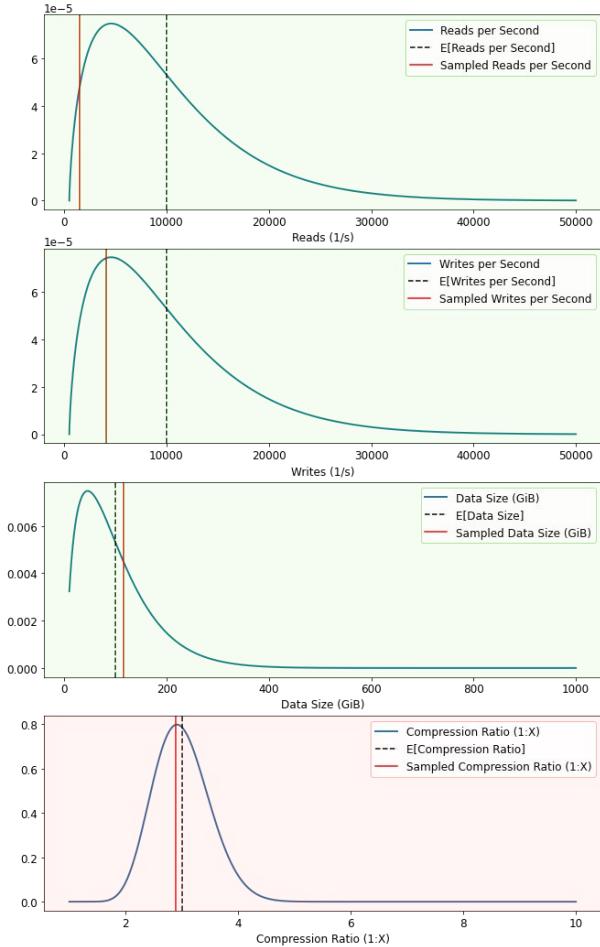


A cheap but regretful world

6 r5d.large

\$2,854.34

Underprovisioned!

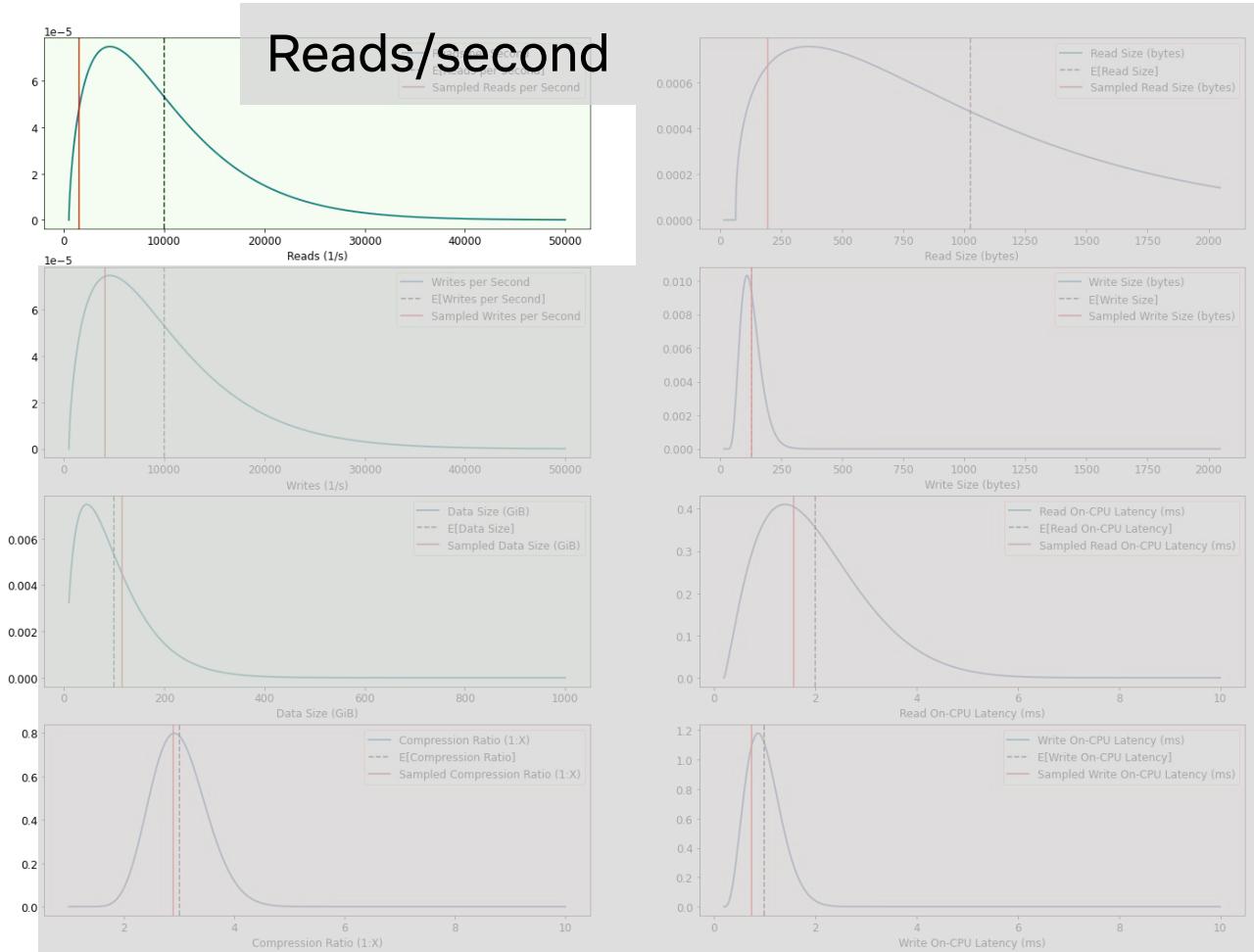


**A cheap but
regretful world**

6 r5d.large

\$2,854.34

Underprovisioned!



Least Regret: A Different Requirement

```
desires_footprint = CapacityDesires(  
    # This service is critical to the business  
    service_tier=1,  
    query_pattern=QueryPattern(  
        estimated_read_per_second=Interval(  
            low=1_000, mid=10_000, high=100_000, confidence=0.98  
        ),  
        estimated_write_per_second=Interval(  
            low=10_000, mid=100_000, high=1_000_000, confidence=0.98  
        ),  
    ),  
    # Not sure how much data, but we think it'll be around 10 TiB  
    data_shape=DataShape(  
        estimated_state_size_gib=Interval(  
            low=1_000, mid=10_000, high=100_000, confidence=0.98),  
    ),  
)
```

Least Regret: A Different Requirement

Least Regret Choice:

48 i3en.xlarge costing 73652.57

A lot more variability based on input!

But we still picked 48
i3en.xlarge 165/1024 times

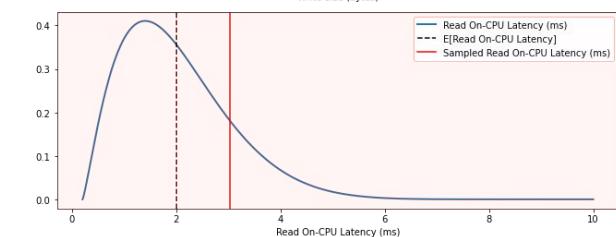
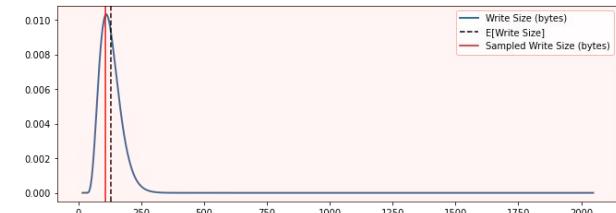
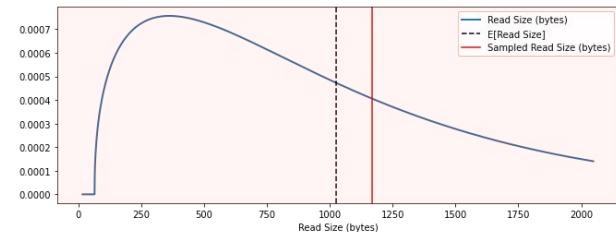
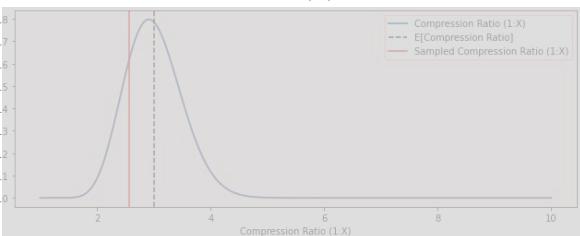
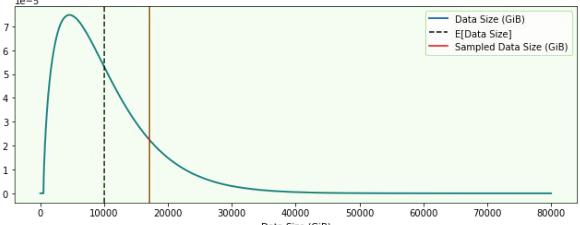
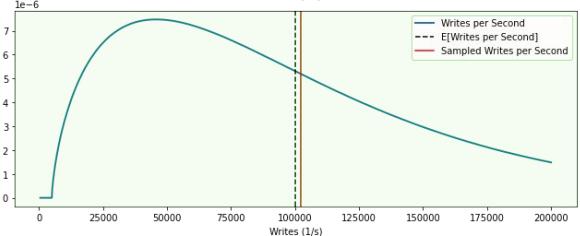
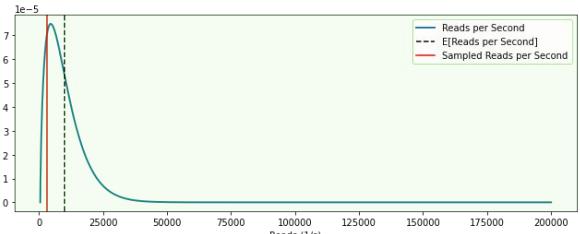
```
All Choices
-----
{' 6 i3en.2xlarge': 13,
' 6 i3en.3xlarge': 14,
' 6 i3en.xlarge': 2,
' 6 m5.8xlarge': 6,
' 6 m5d.4xlarge': 2,
' 6 m5d.8xlarge': 17,
' 6 r5.4xlarge': 3,
' 6 r5.8xlarge': 7,
' 12 i3en.2xlarge': 81,
' 12 i3en.3xlarge': 43,
' 12 i3en.xlarge': 17,
' 12 m5.4xlarge': 1,
' 12 m5.8xlarge': 7,
' 12 m5d.2xlarge': 1,
' 12 r5.2xlarge': 4,
' 12 r5.4xlarge': 16,
' 12 r5.8xlarge': 6,
' 24 r5.large': 1,
' 24 r5.xlarge': 3,
' 24 i3.2xlarge': 3,
' 24 i3en.2xlarge': 144,
' 24 i3en.3xlarge': 27,
' 24 i3en.xlarge': 162,
' 24 m5.2xlarge': 5,
' 24 m5.4xlarge': 4,
' 24 m5.8xlarge': 4,
' 24 r5.2xlarge': 9,
' 24 r5.4xlarge': 12,
' 48 r5.large': 2,
' 48 i3.xlarge': 5,
' 48 m5.xlarge': 16,
' 48 r5.xlarge': 5,
' 48 i3.2xlarge': 2,
' 48 i3en.2xlarge': 43,
' 48 i3en.3xlarge': 4,
' 48 i3en.xlarge': 165,
' 48 m5.2xlarge': 18,
' 48 m5.4xlarge': 1,
' 48 m5.8xlarge': 1,
' 48 m5d.xlarge': 1,
' 48 r5.2xlarge': 9,
' 48 r5.4xlarge': 2,
' 96 r5.large': 34,
' 96 i3.xlarge': 7,
' 96 m5.xlarge': 30,
' 96 r5.xlarge': 64,
' 96 i3en.2xlarge': 1,
' 96 i3en.xlarge': 33,
' 96 m5.2xlarge': 17,
' 96 r5.2xlarge': 8,
' 96 r5.4xlarge': 1,
' 96 r5.8xlarge': 1}
```

Least Regret

48 i3en.xlarge

\$73,652.57

Good amount of disk

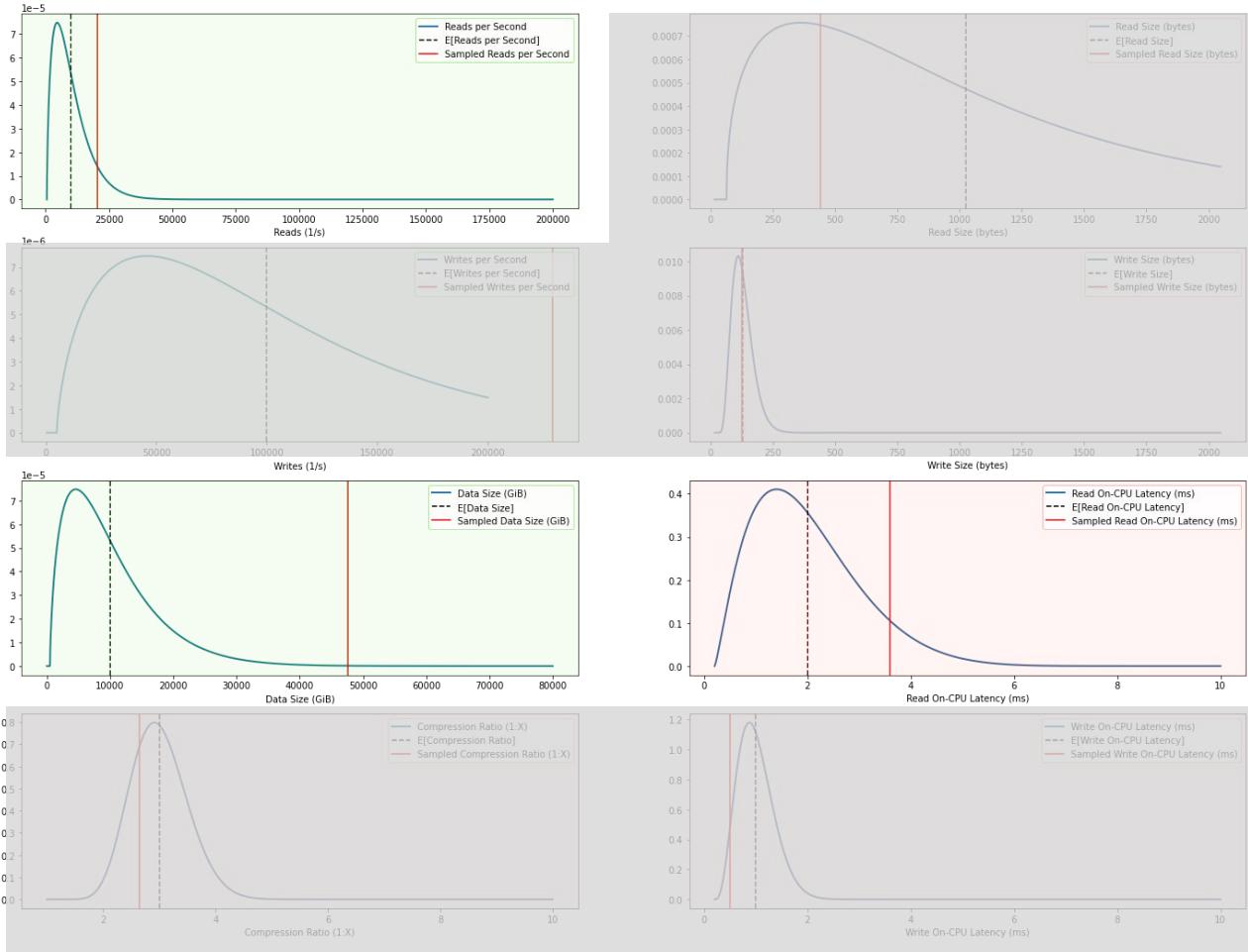


Least Regret

**96 r5.8xlarge
with 1.2TiB gp2**

\$646,309.93

Too much money!



Monitoring

How do you know
you've run out of
capacity?

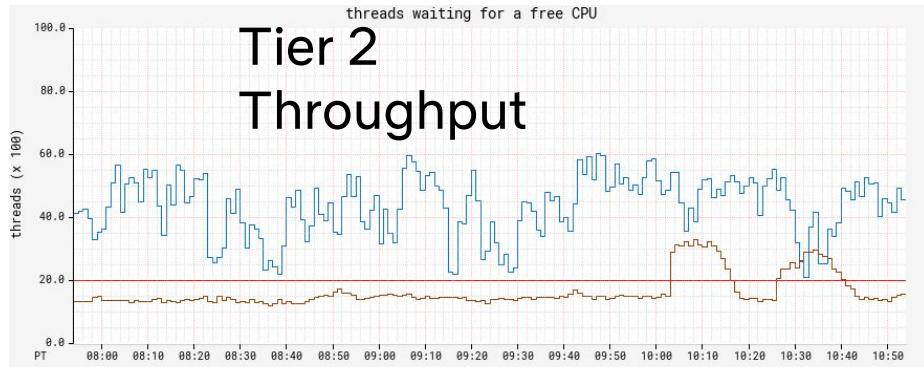
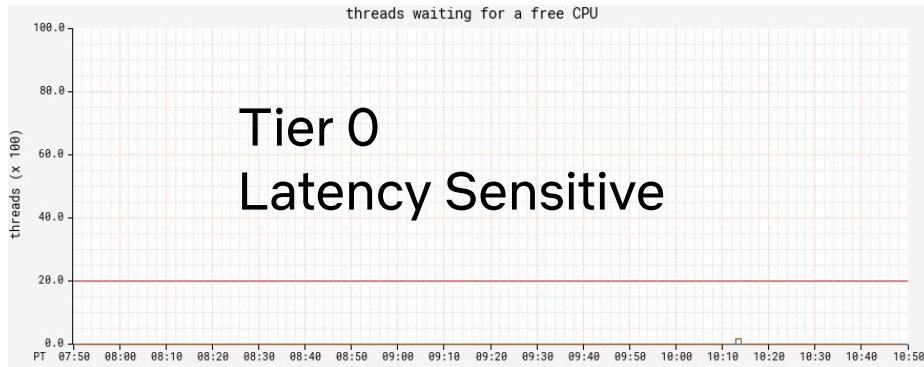


CPU

Measure /proc/schedstat

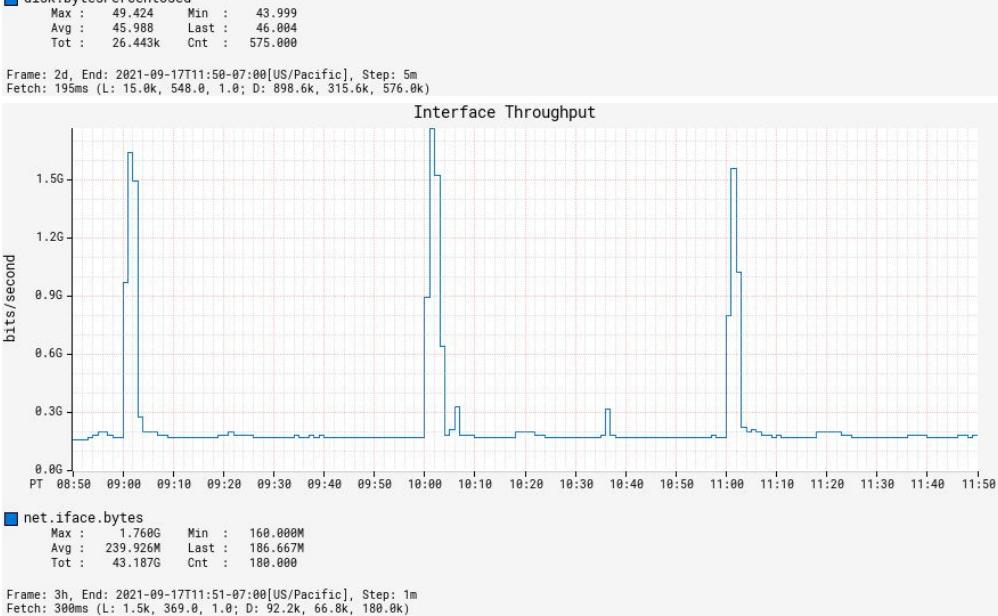
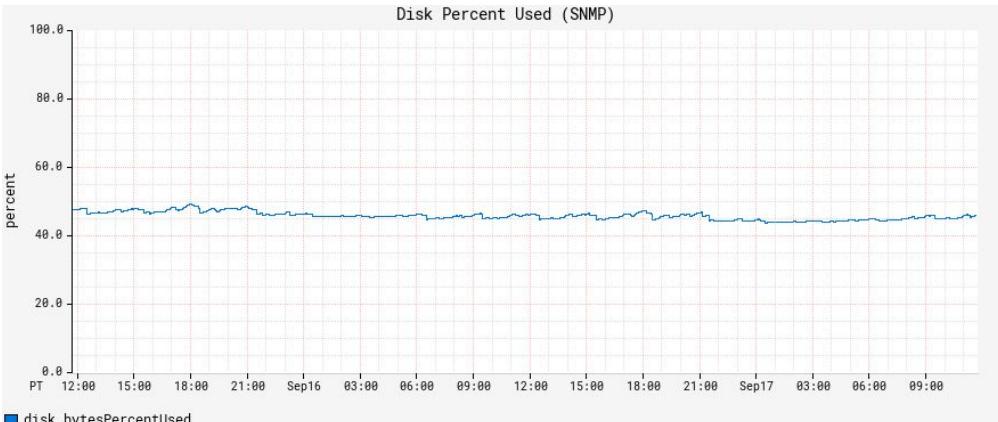
"would additional CPUs help me"

```
def gather_metric():
    # scale time spent in the scheduler by this factor
    schedstat_lines = open('/proc/schedstat').readlines()
    delays = [
        int(i.split(' ')[8]) for i
        in schedstat_lines if i.startswith('cpu')
    ]
    delays = delays or [0]
    return sum(delays) / float(len(delays))
```



Disk Network

Basic utilization metrics suffice



RAM

Page Cache

Use read IO metrics

Or bpf if you're fancy
([cachestat](#))

JVM/Write Buffer

Major garbage collection frequency > ~10 minutes

Flush frequency > ~4 minutes

Monitoring Your Choices

Buy more of whatever you ran out of.

Need more memory?
M5 -> R5

Need more network?
R5 -> R5n

Conclusion

Understanding Hardware

We measured, priced and imposed lifecycle on our hardware

Capacity Planning

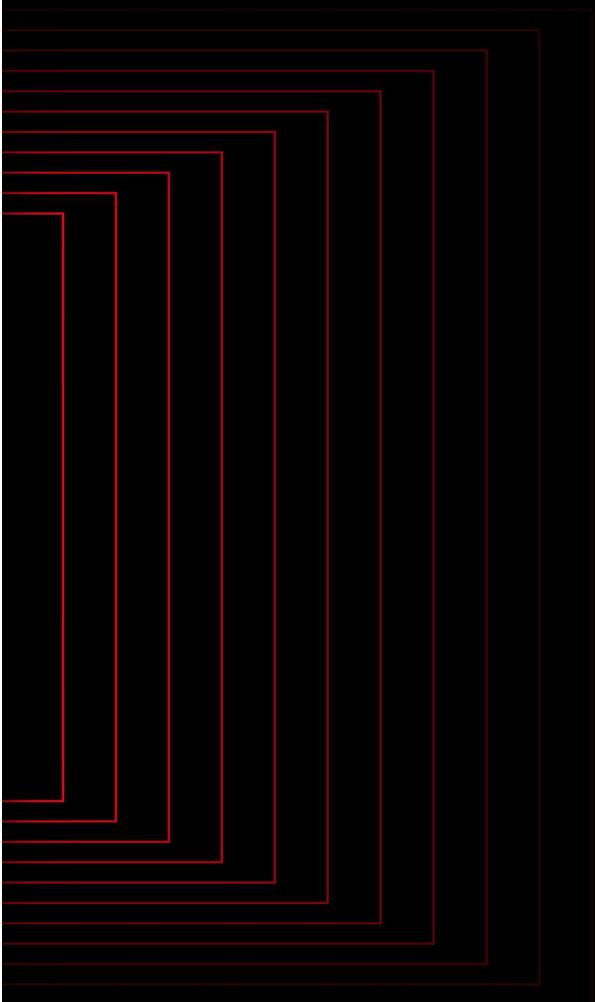
Apply queueing theory with anger
Simulate worlds, pick least regretful

Monitoring your Choices

Buy more of what you need

Questions

N



Demo

