How Netflix Provisions Optimal Cloud Deployments of Cassandra

Joey Lynch
Senior Software Engineer
Cloud Data Engineering - Netflix

Speaker

Joey Lynch

Senior Software Engineer
Cloud Data Engineering at Netflix

Database shepherd and data wrangler

https://jolynch.github.io/

Show me the Code!

Service Capacity Modeling

Build passing

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

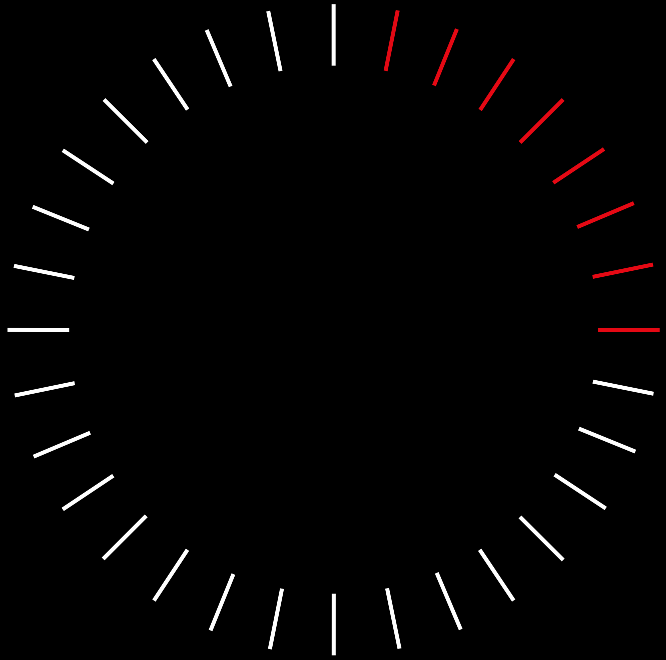
$$M(D, H, PL) \rightarrow C$$

$$\begin{aligned}
M &= \text{Workload Capacity Model} \\
D &= \text{User Desire} \\
H &= \text{Hardware Profile} \\
PL &= \text{Current Pricing and Lifecycle} \\
C &= \text{Candidate Cluster}
\end{aligned}$$

Under-
standing

Hardware

There are a lot of computers
… and they cost money

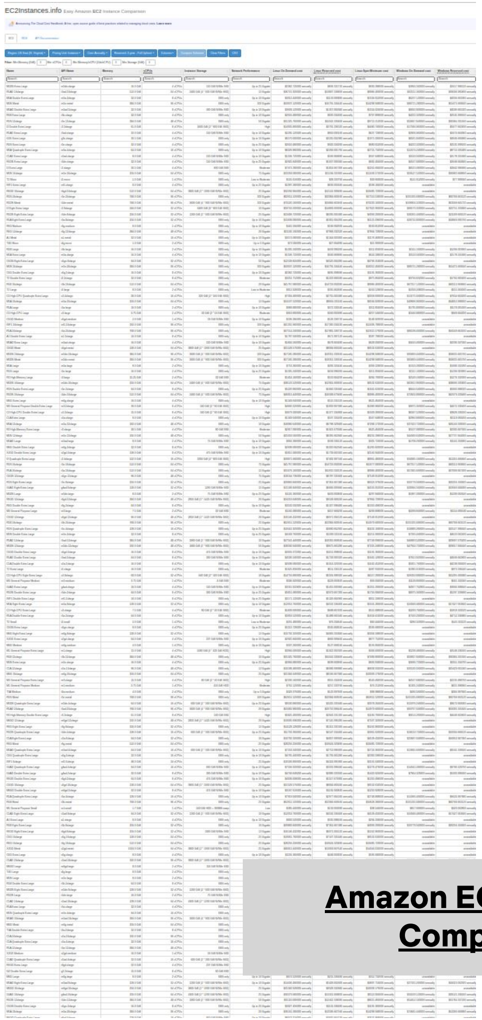


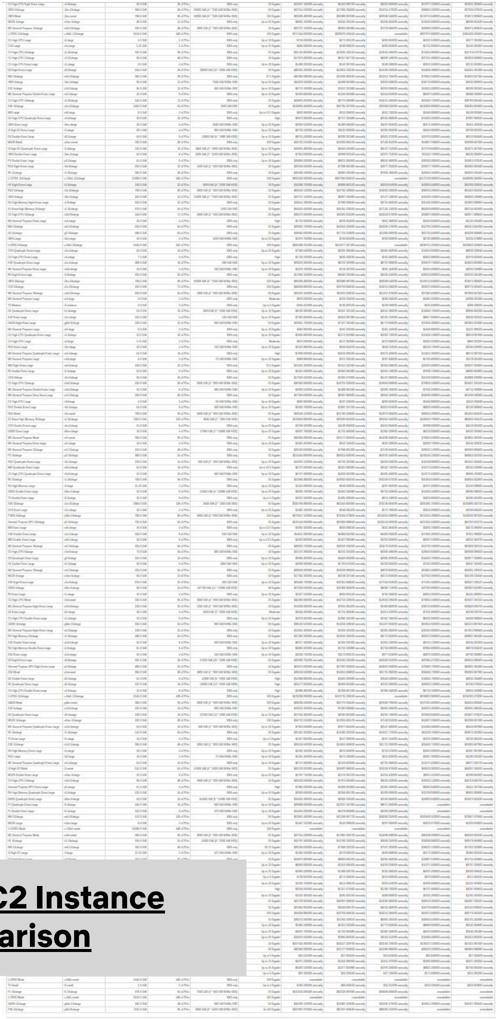
Capacity Planning 101

$$\mathtt{M}(\mathtt{D},\ \mathrm{H},\ \mathrm{PL}) \rightarrow \mathtt{C}$$

M = Workload Capacity Model
D = User Desire
H = Hardware Profile
PL = Current Pricing and Lifecycle
C = Candidate Cluster

Hardware





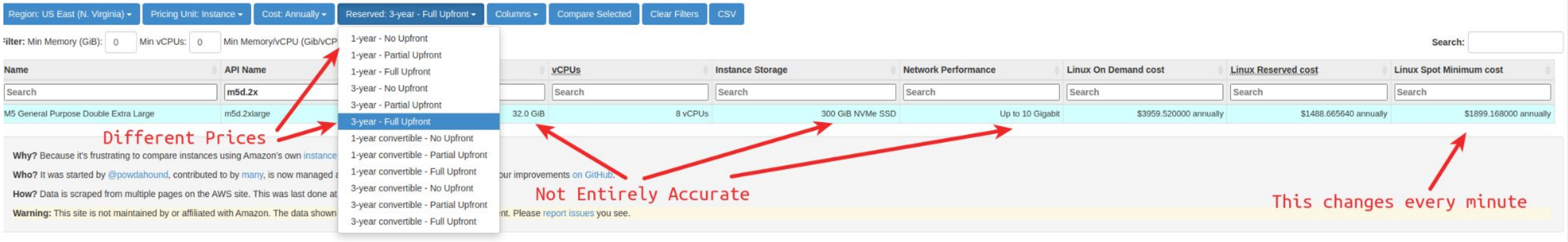
[Amazon EC2 Instance Comparison]

Hundreds of choices

With confusing names

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

Hardware



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

Problem

$$\mathtt{M}(\mathtt{D}, \mathrm{H}, \mathrm{PL}) \rightarrow \mathtt{C}$$

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

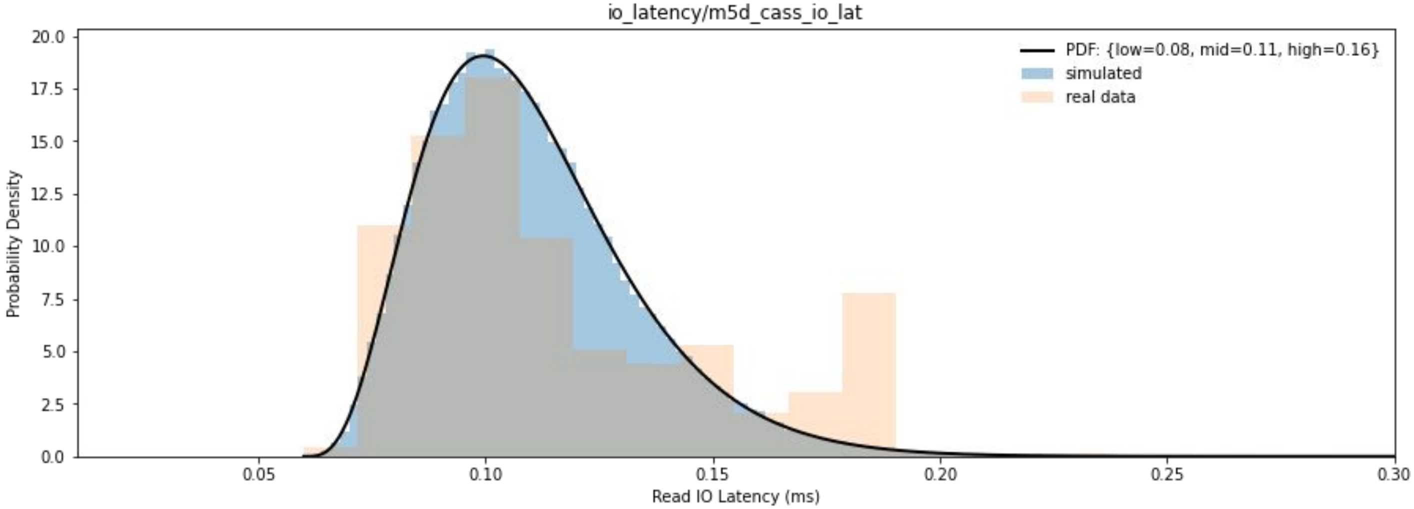
Solution!

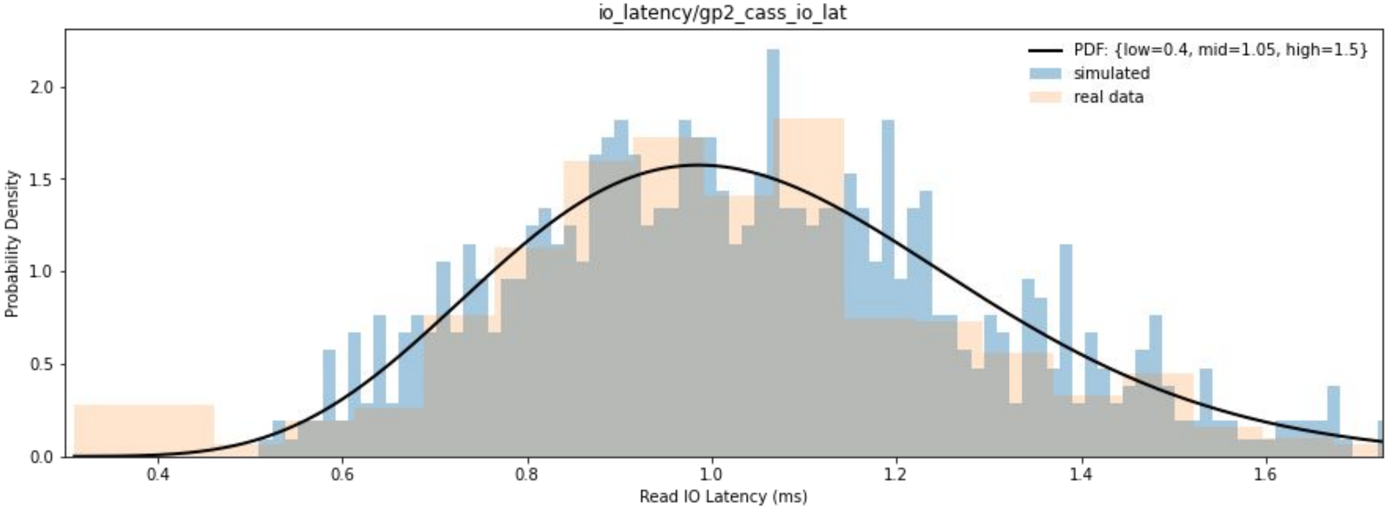
For a concrete example let's model a m5d drive which we model with an io latency distribution. Data for comparision comes from using `biosnoop` and `histogram.py` on a Cassandra server (the threads that are servicing reads are from the SharedPool).

```
$ 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%)
```

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/notebooks/drive_latency.ipynb

Solution!

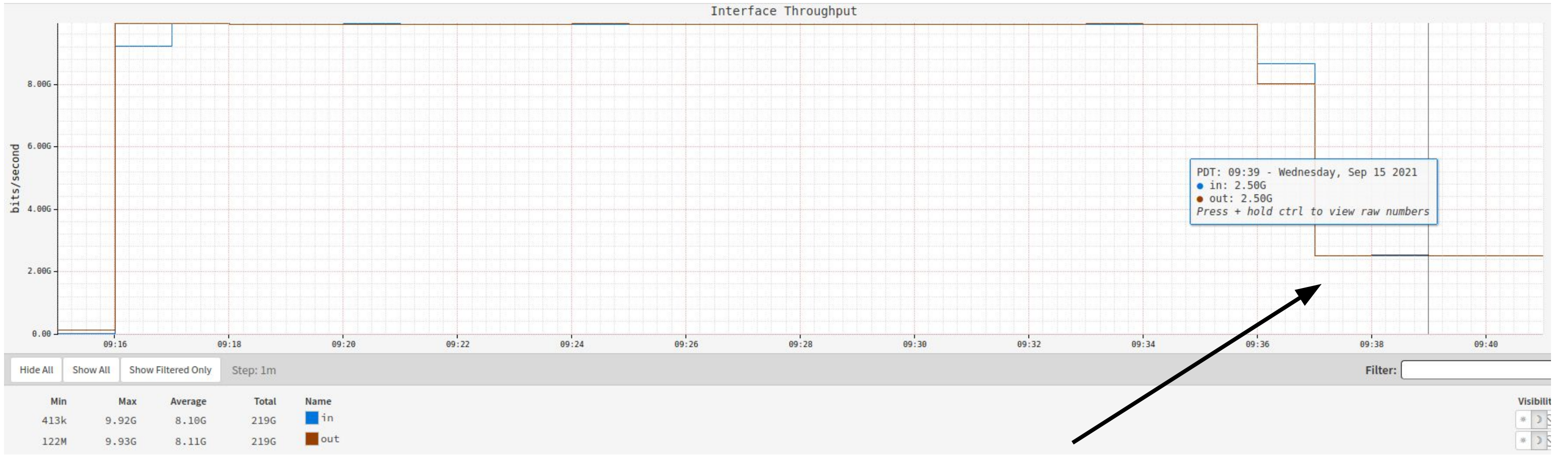

io_latency/m5d_cass_io_lat
PDF: {low=0.08, mid=0.11, high=0.16}
simulated
real data
Probability Density
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.05
0.10
0.15
0.20
0.25
0.30
Read IO Latency (ms)


io_latency/gp2_cass_io_lat
PDF: {low=0.4, mid=1.05, high=1.5}
simulated
real data
Probability Density
2.0
1.5
1.0
0.5
0.0
0.4
0.6
0.8
1.0
1.2
1.4
1.6
Read IO Latency (ms)

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": "0.0000004"
  }
}
```

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

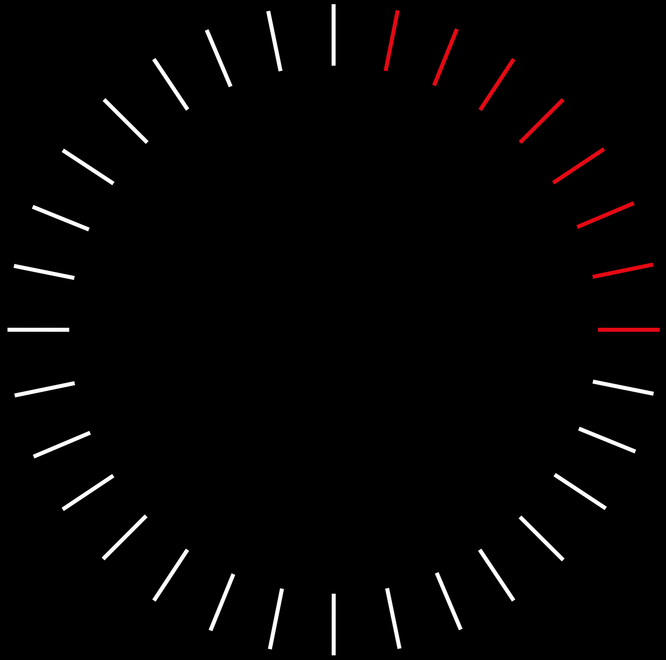
Capacity Planning 201

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

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

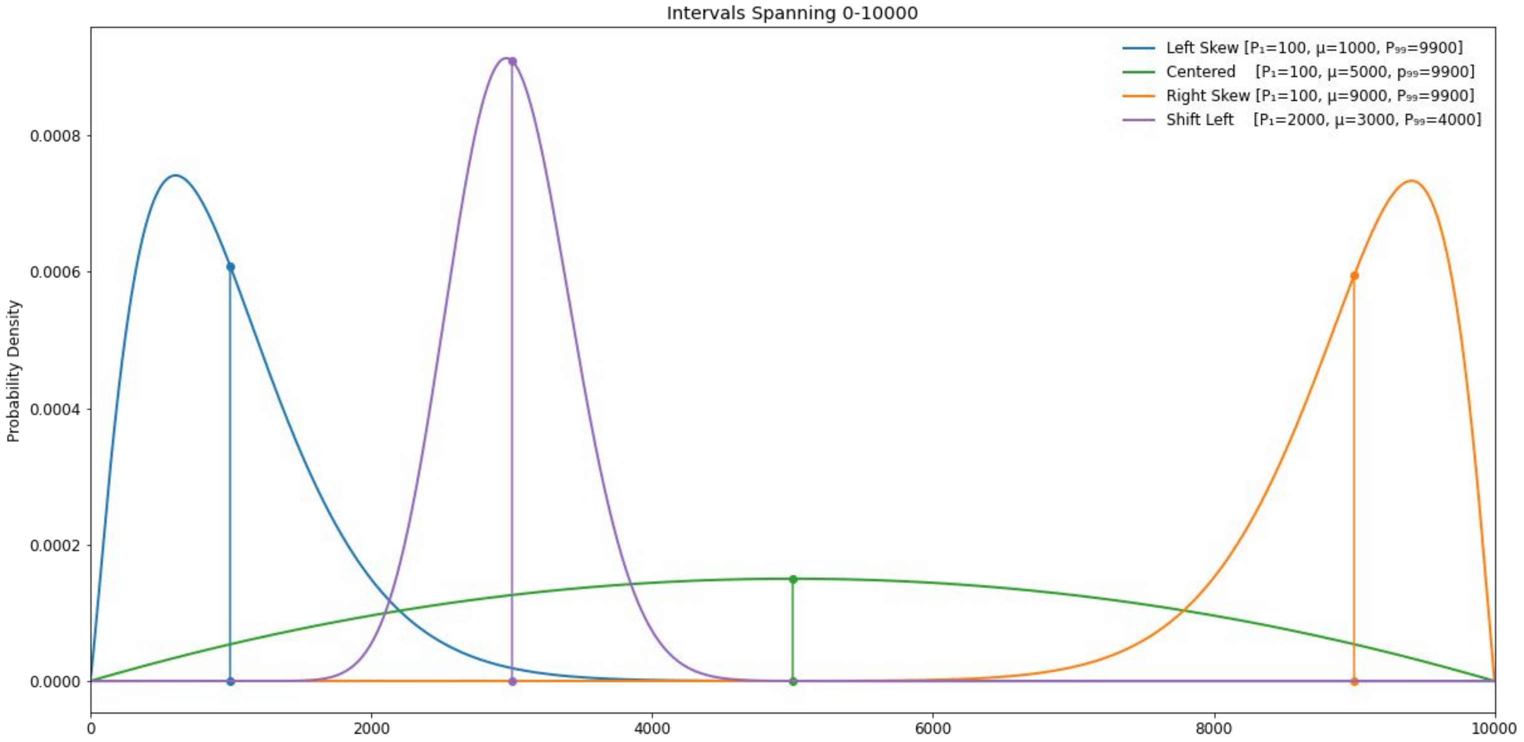
The user probably knows how much CPU, RAM, Network, Disk space, and Disk IOs they need

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

Service Tier

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
)
```

How is it queried?

- read/write
- sizing
- latency

Provide defaults from the **model**

Inputs are **Intervals**

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

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
)
```

How is the data shaped?

footprint

durability

Provide defaults from the **model**

Inputs are **Intervals**

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

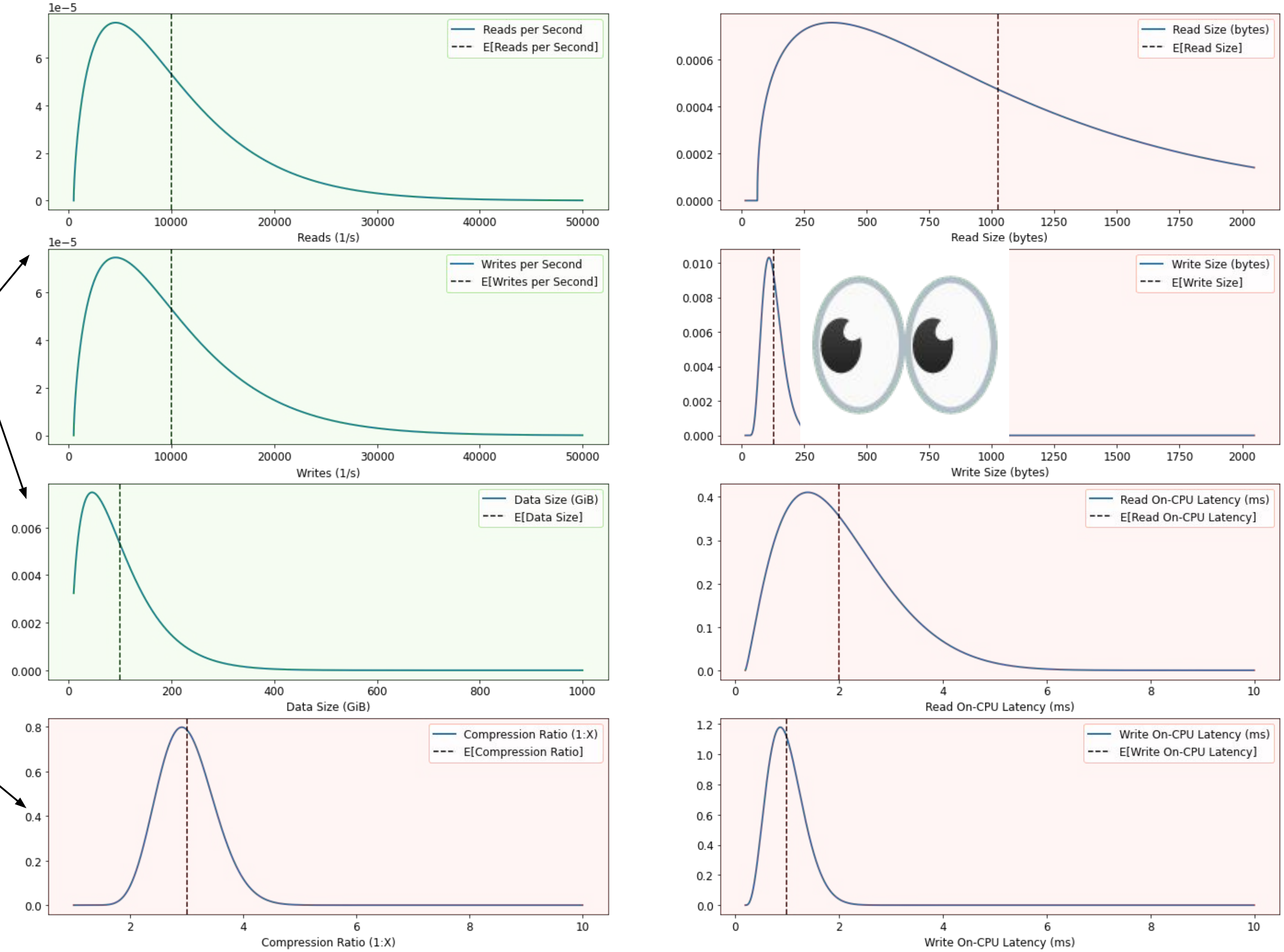
Intervals

```python
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

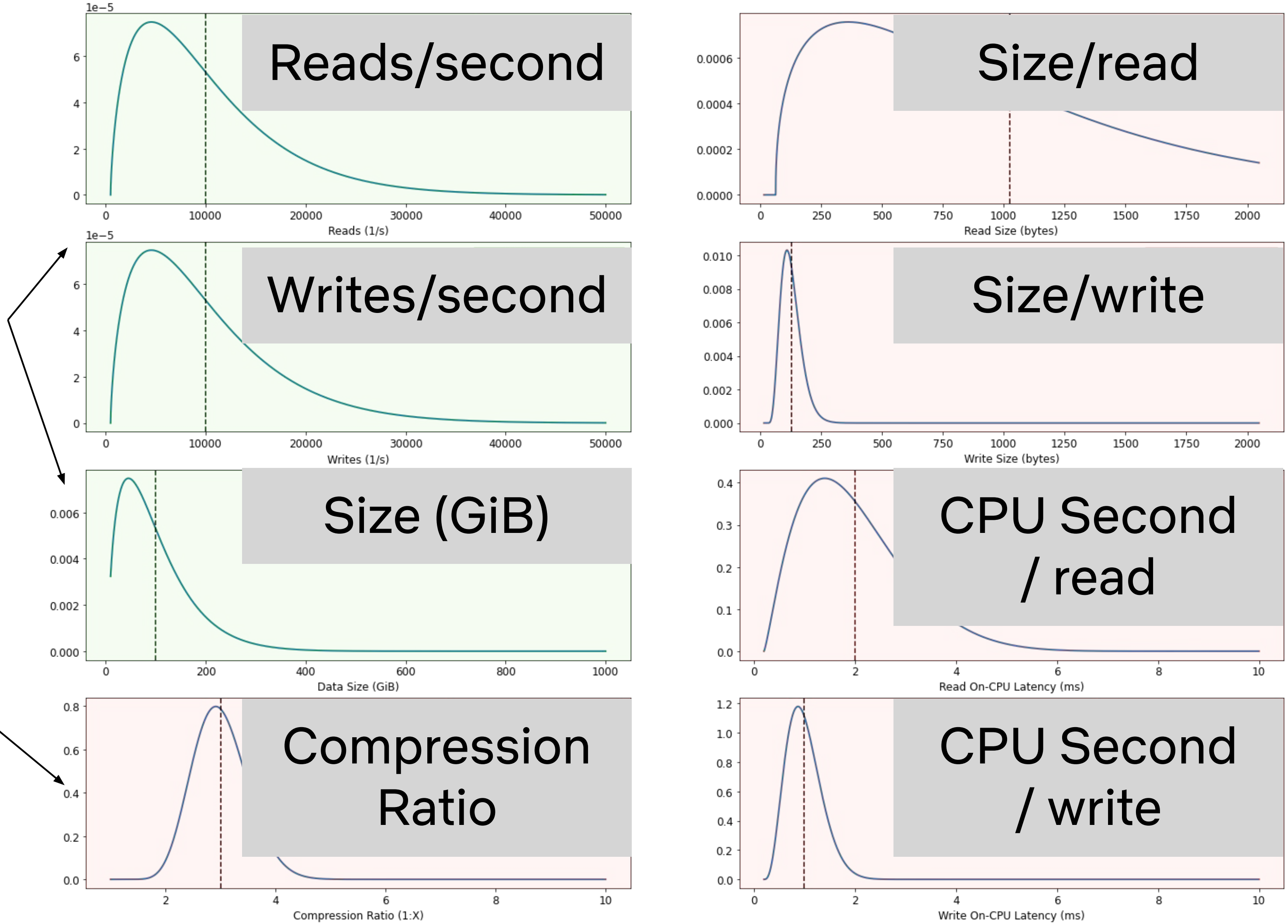
From the Model



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"

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

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

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

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

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

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

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/models/common.py#L24-L66

CPU

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,

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

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

M. Harchol-Balter, Performance Modeling and Design of Computer Systems: Queueing Theory in Action. (google books)

CPU

Reads/second
Reads (1/s)
Writes/second
Writes (1/s)
Read Size (bytes)
E[Read Size]
Write Size (bytes)
E[Write Size]
Data Size (GiB)
E[Data Size]
Compression Ratio (1:X)
E[Compression Ratio]
CPU Second / read
Read On-CPU Latency (ms)
CPU Second / write
Write On-CPU Latency (ms)

Network

For simple case it's easy

Tricky in complex case...

We have to know Consistency Level and Replication Factor

$$\text{let } \mu_r = \frac{\text{bytes}}{\text{read}} \qquad \text{let } \mu_w = \frac{\text{bytes}}{\text{write}}$$

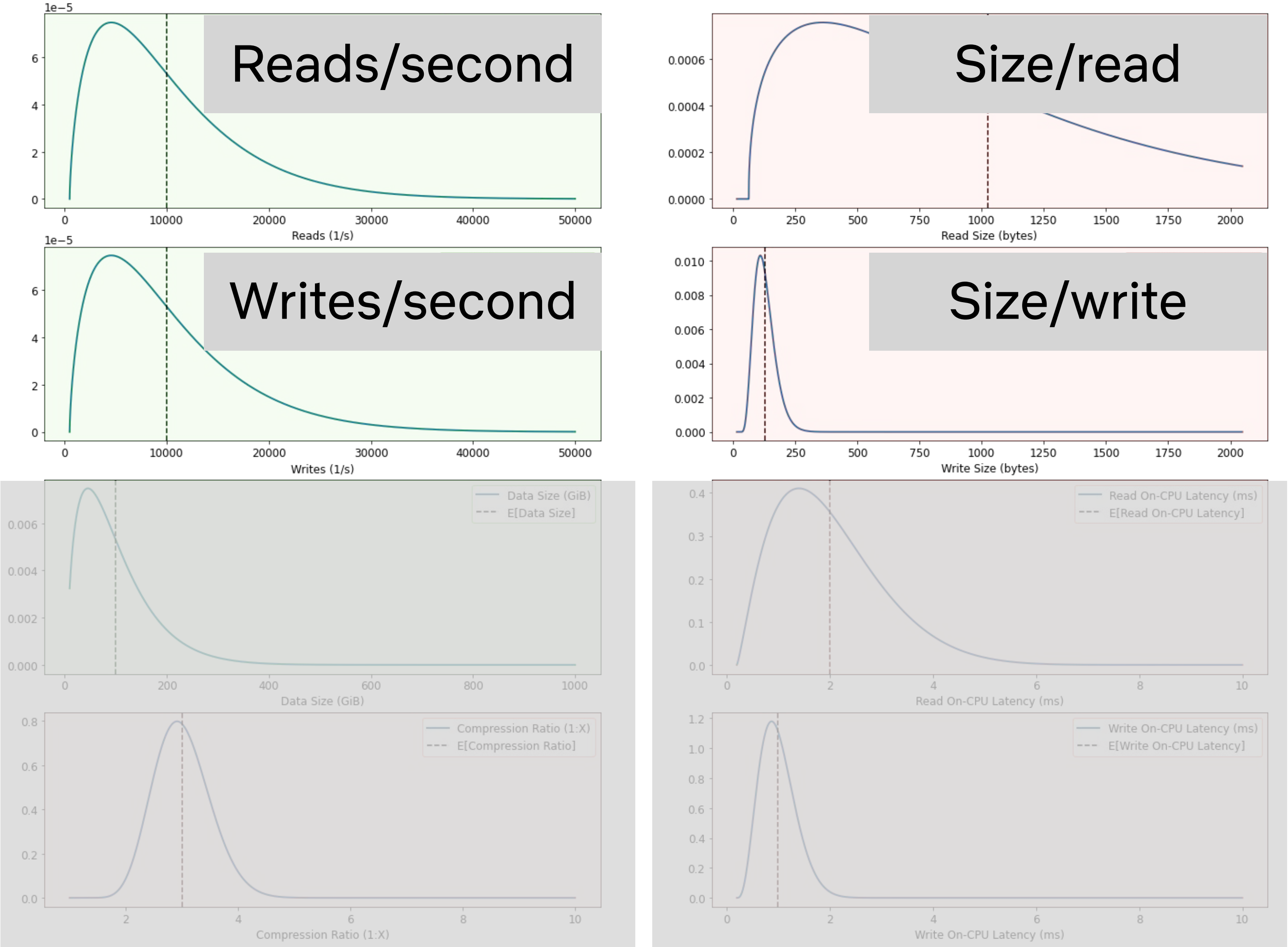
$$\text{let } \lambda_r = \frac{\text{read}}{\text{second}} \qquad \text{let } \lambda_w = \frac{\text{write}}{\text{second}}$$

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

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

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/models/common.py#L69-L82

Network



Reads/second
Reads (1/s)
Writes/second
Writes (1/s)
Size/read
Read Size (bytes)
Size/write
Write Size (bytes)
Data Size (GiB)
E[Data Size]
Read On-CPU Latency (ms)
E[Read On-CPU Latency]
Compression Ratio (1:X)
E[Compression Ratio]
Write On-CPU Latency (ms)
E[Write On-CPU Latency]

Disk

Compaction strategy and Compression matter

Tricky: Remember network drives must be sized for IO

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

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

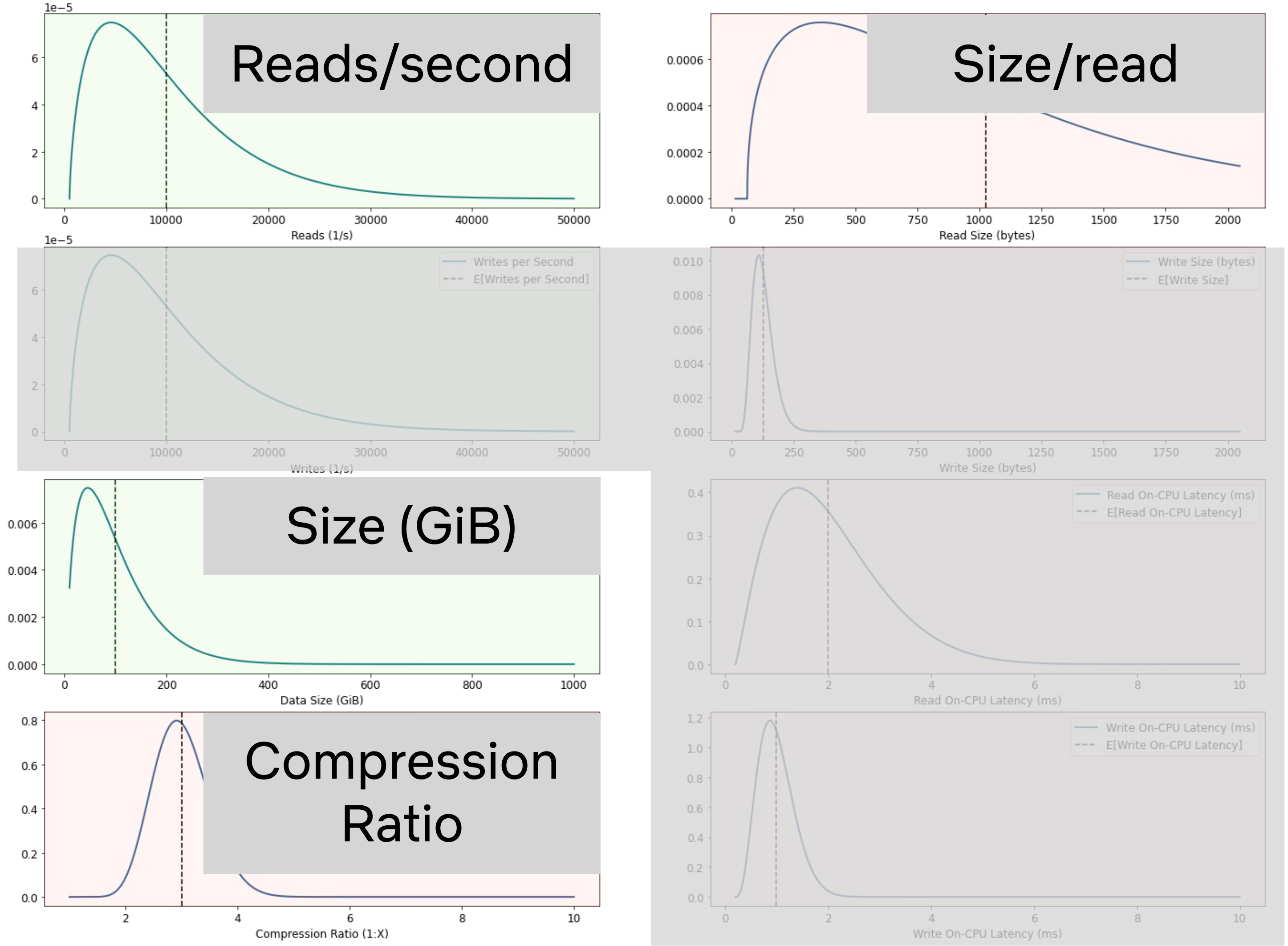
$$\text{Epem}_{\text{node}} = \text{size}_{\text{node}}$$

$$OR$$

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

https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/models/org/netflix/cassandra.py#L84-L88

Disk


Reads/second
Size/read
Size (GiB)
Compression
Ratio
Reads (1/s)
Read Size (bytes)
Writes per Second
E[Writes per Second]
Writes (1/s)
Write Size (bytes)
E[Write Size]
Write Size (bytes)
Data Size (GiB)
Read On-CPU Latency (ms)
E[Read On-CPU Latency]
Read On-CPU Latency (ms)
Compression Ratio (1:X)
Write On-CPU Latency (ms)
E[Write On-CPU Latency]
Write On-CPU Latency (ms)

Memory

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

Tricky: This depends on the number of nodes.

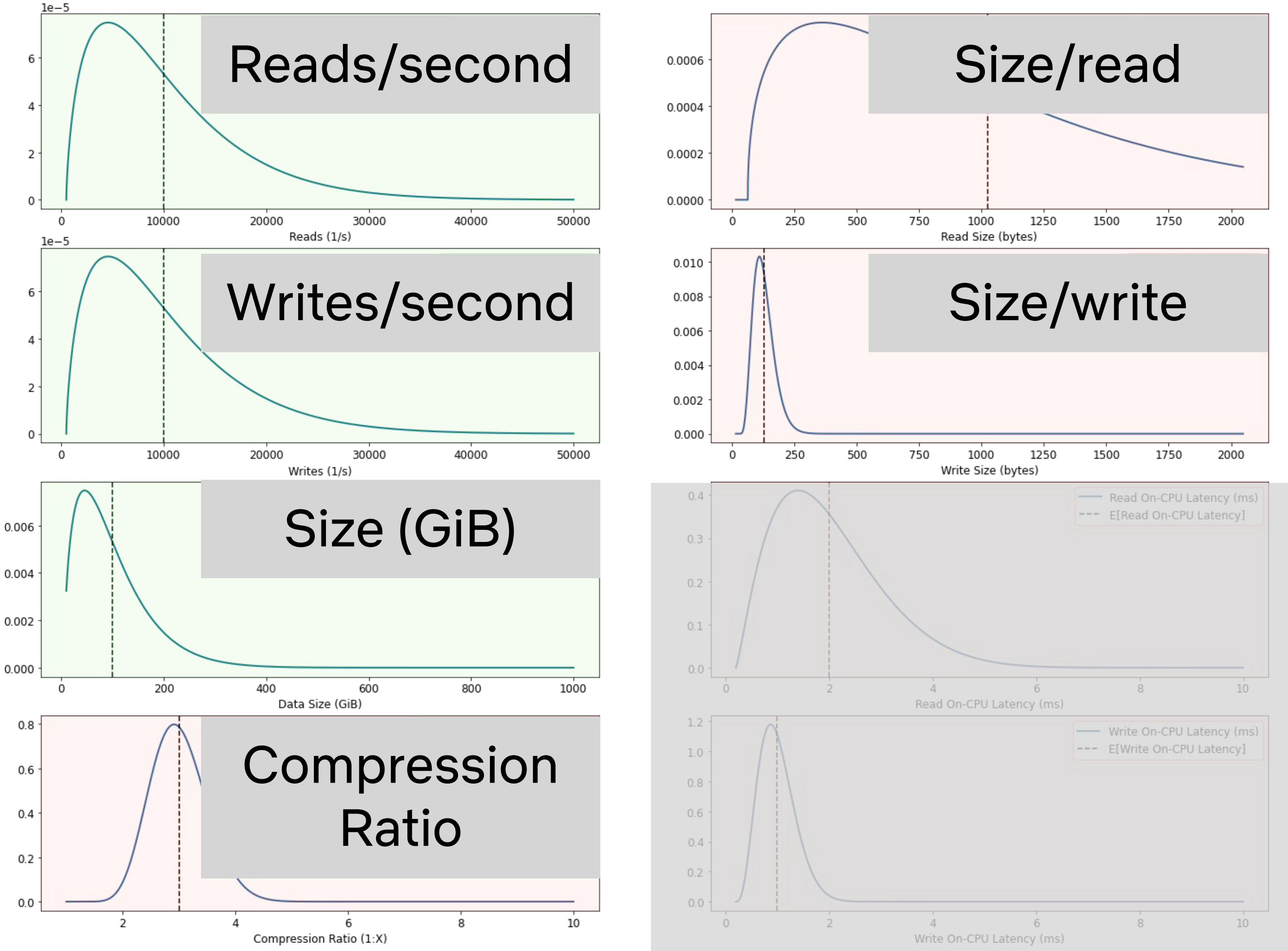
$$\mathrm{RAM}_{read} = f(\text{data size, working set})$$
$$\mathrm{RAM}_{write} = f(\text{write BW, compaction})$$

$$\mathrm{RAM}_{JVM} = f(\text{write BW, read BW})$$
$$\mathrm{RAM}_{system} = f(\text{sidecars, kernel})$$

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

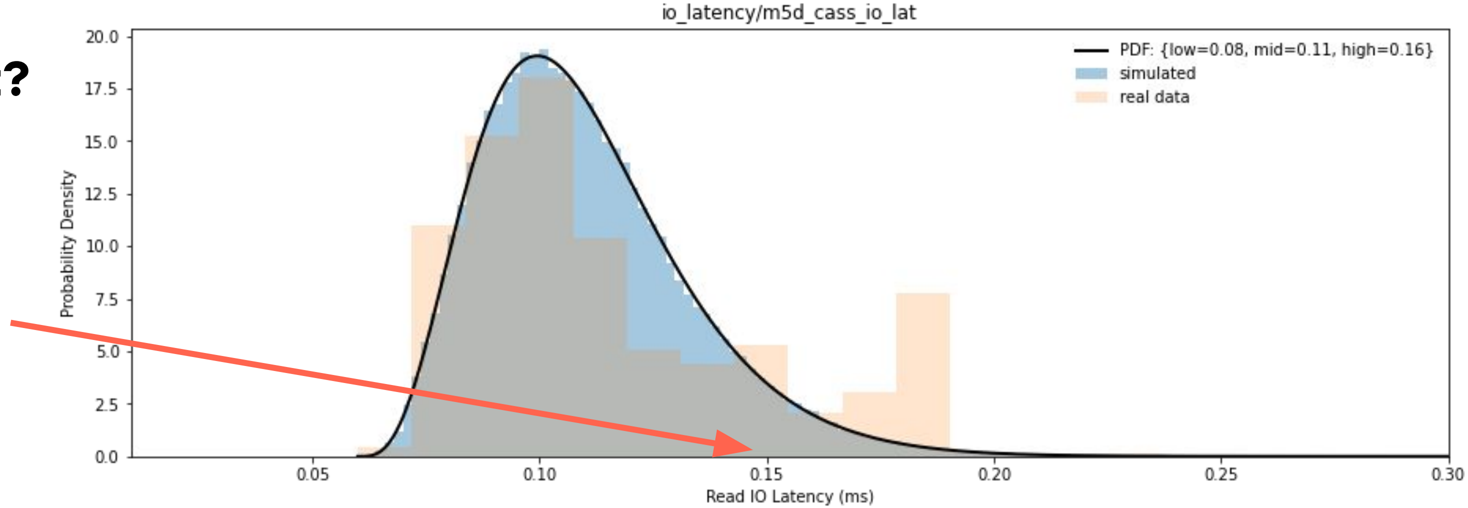
https://github.com/Netflix-Skunkworks/service-capacity-modeling/blob/main/service_capacity_modeling/models/org/netflix/cassandra.py#L42-L157

Memory

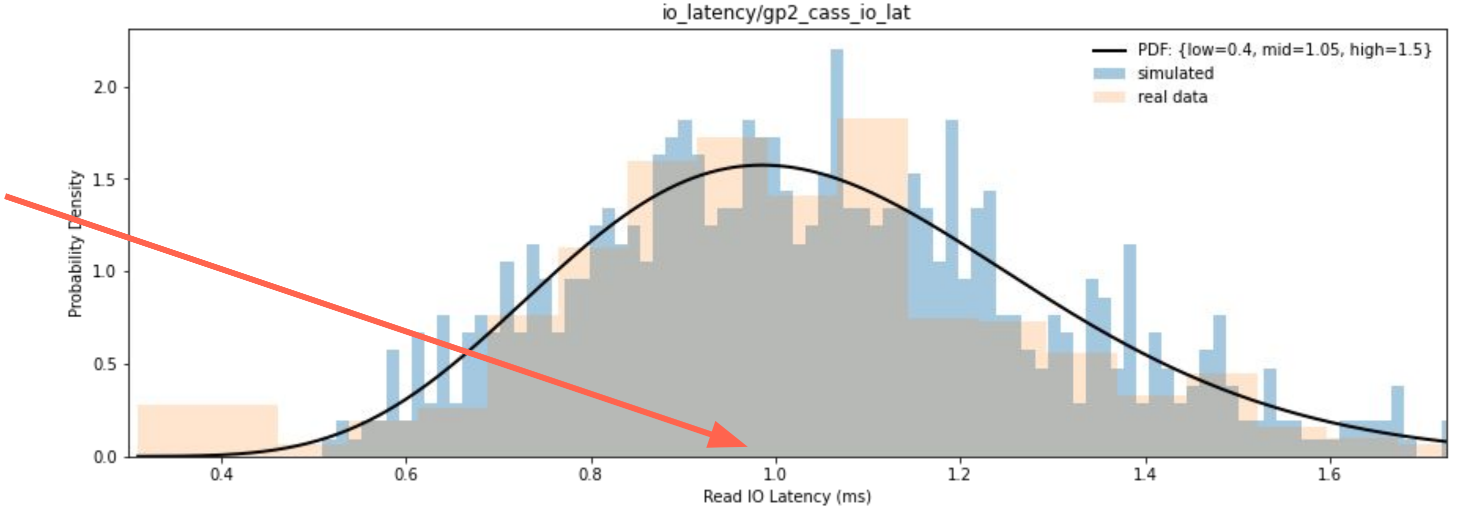

Reads/second
Reads (1/s)
Size/read
Read Size (bytes)
Writes/second
Writes (1/s)
Size/write
Write Size (bytes)
Size (GiB)
Data Size (GiB)
Read On-CPU Latency (ms)
E[Read On-CPU Latency]
Compression Ratio
Compression Ratio (1:X)
Write On-CPU Latency (ms)
E[Write On-CPU Latency]

Working Set?

This needs very little RAM

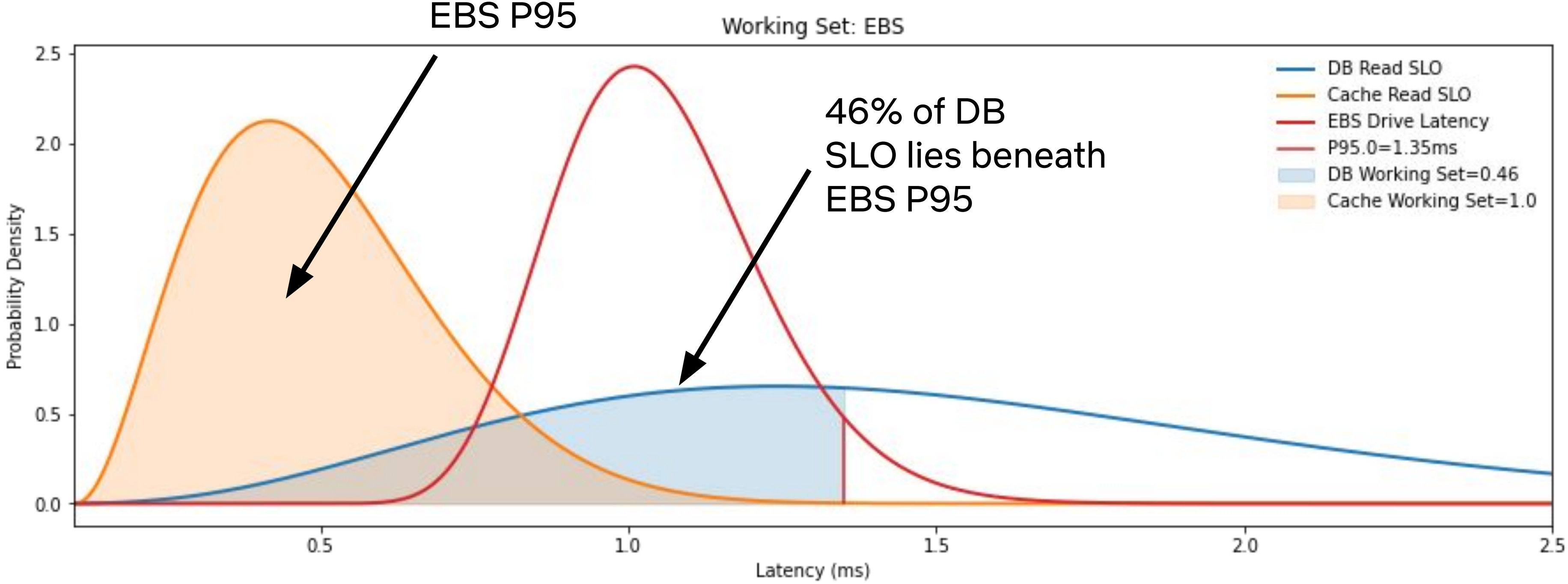


This needs more RAM



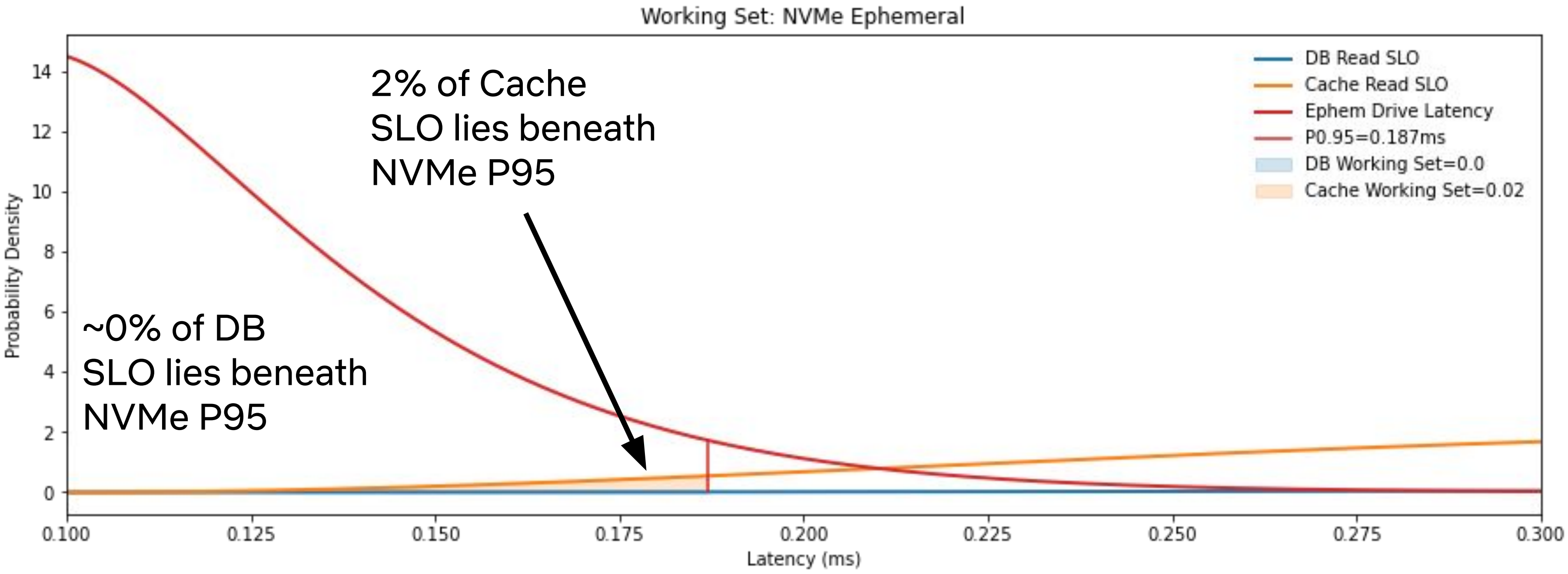
Working Set

>99% of Cache SLO lies beneath EBS P95



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

$$\forall \mathtt{H}(\mathtt{M}_{\text{cassandra}}(\mathtt{D},\ \mathtt{H},\ \mathtt{P})) \longrightarrow \mathtt{C}_{\mathtt{H}}$$

choose

$$C = \text{argmin}_{\mathtt{H}}(\mathit{cost}(C_H))$$

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

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

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

https://pixabay.com/illustrations/multiverse-parallel-universe-6508796/

Capacity Planning

Take it to 11

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

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)

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

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

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

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

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

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

$$\text{regret}(\mathtt{W}_1 \text{ in } \mathtt{W}_2)_{\$} = 1.25 \times |73,652.57 - 646,309.93|^{1.2} \approx 10M$$

$$\text{regret}(\mathtt{W}_1 \text{ in } \mathtt{W}_2)_{\text{disk}} = 1.10 \times |6,634.0 - 17,941|^{1.05} \approx 20K$$

$$\text{regret}(W_1 \text{ in } W_2) \approx 10 \text{ million dollars (underprovisioned)}$$

World 2 IN World 1

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

We **have** 17,941 GiB

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

We **required** 6,634.0 GiB

$$\text{regret}(\mathtt{W}_2 \text{ in } \mathtt{W}_1)_{\$} = 1.0 \times |73,652.57 - 646,309.93|^{1.2} \approx 8M$$

$$\text{regret}(\mathtt{W}_2 \text{ in } \mathtt{W}_1)_{\text{disk}} = 0.0 \times |6,634.0 - 17,941|^{1.05} = 0K$$

$$\text{regret}(W_2 \text{ in } W_1) \approx 8 \text{ million dollars (overprovisioned)}$$

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

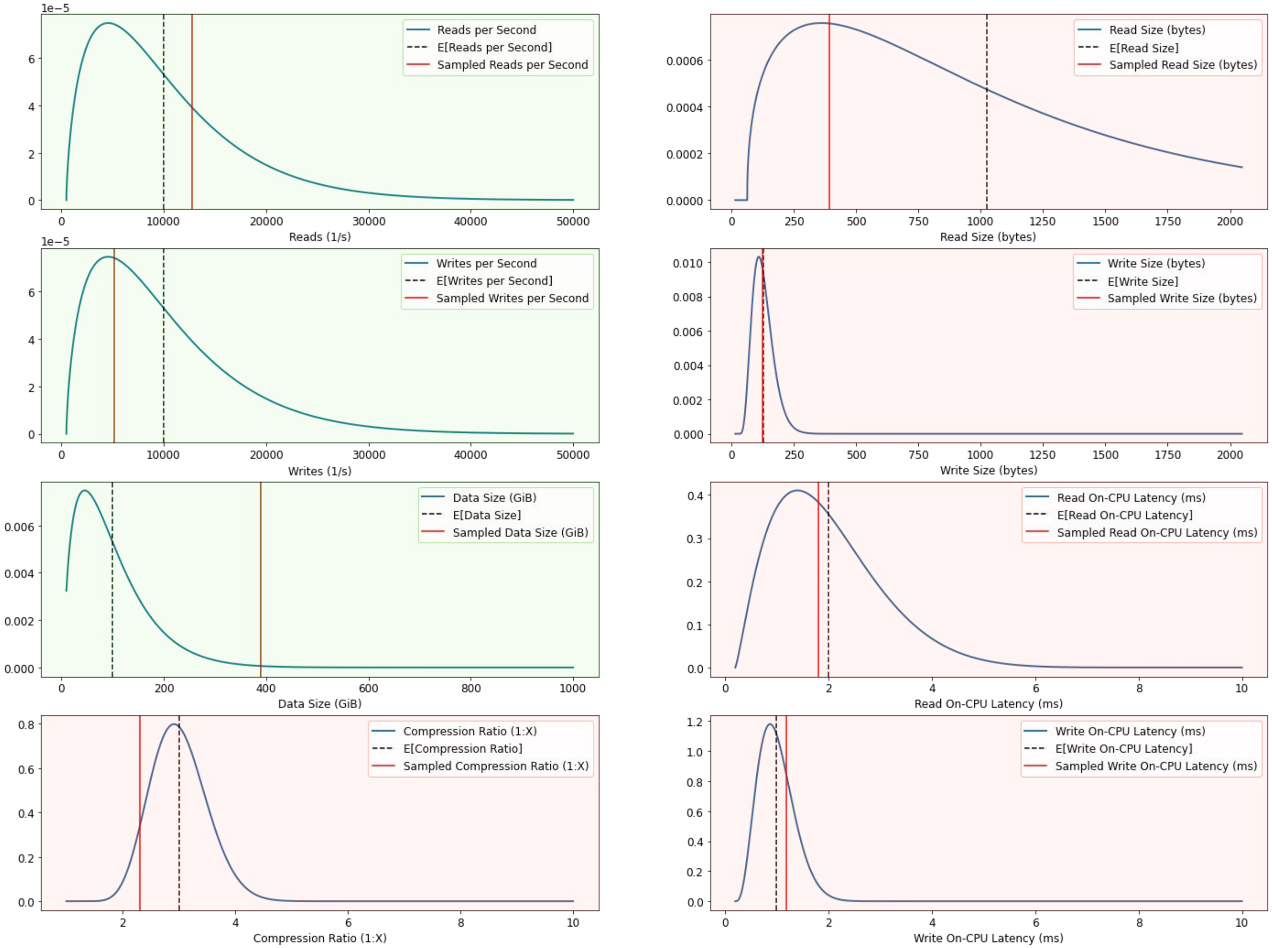
```python
from service_capacity_modeling.capacity_planner import planner
from service_capacity_modeling.models.org import netflix

# 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

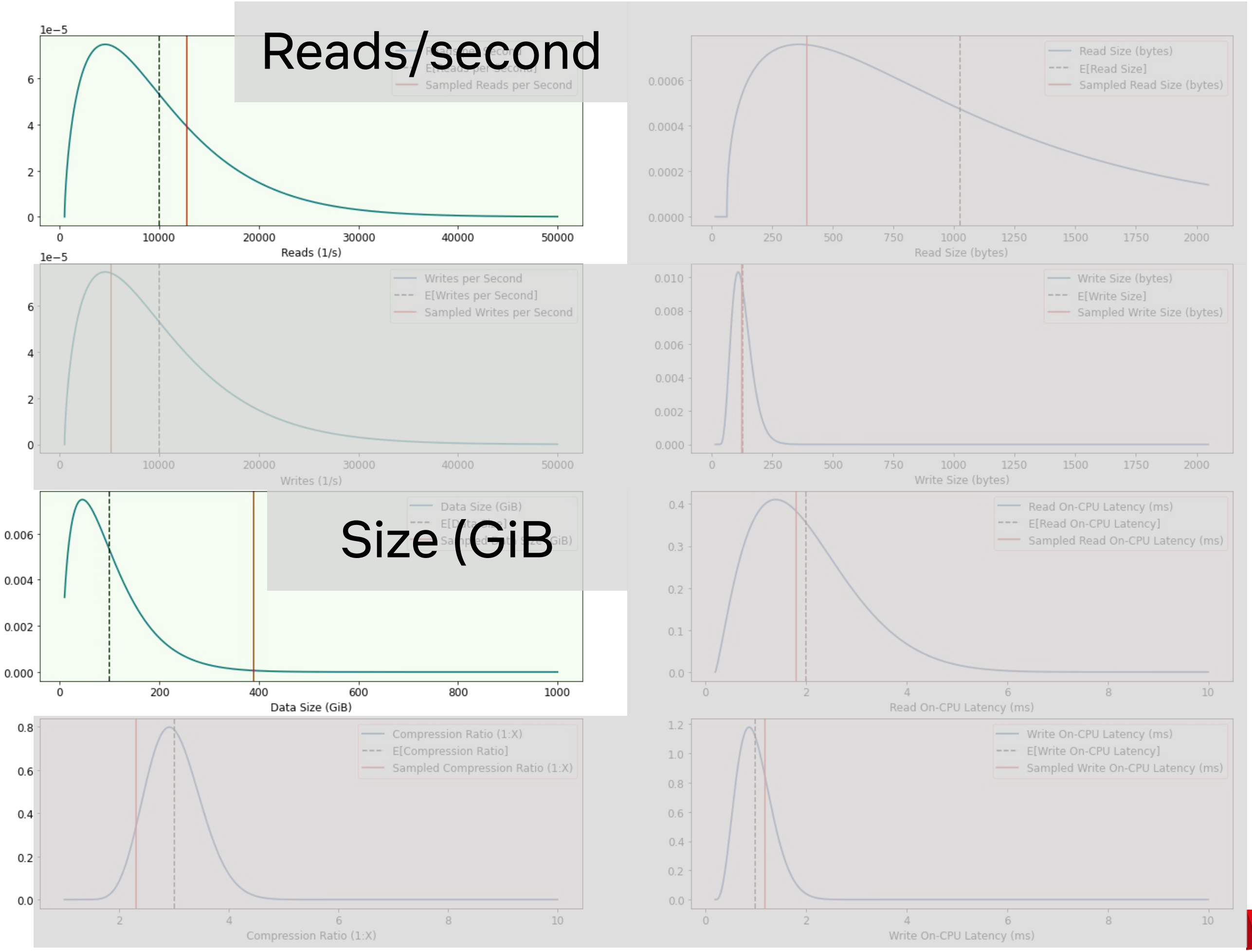
$8,973.94
per year



Least Regret World

12 m5d.xlarge

$8,973.94
per year



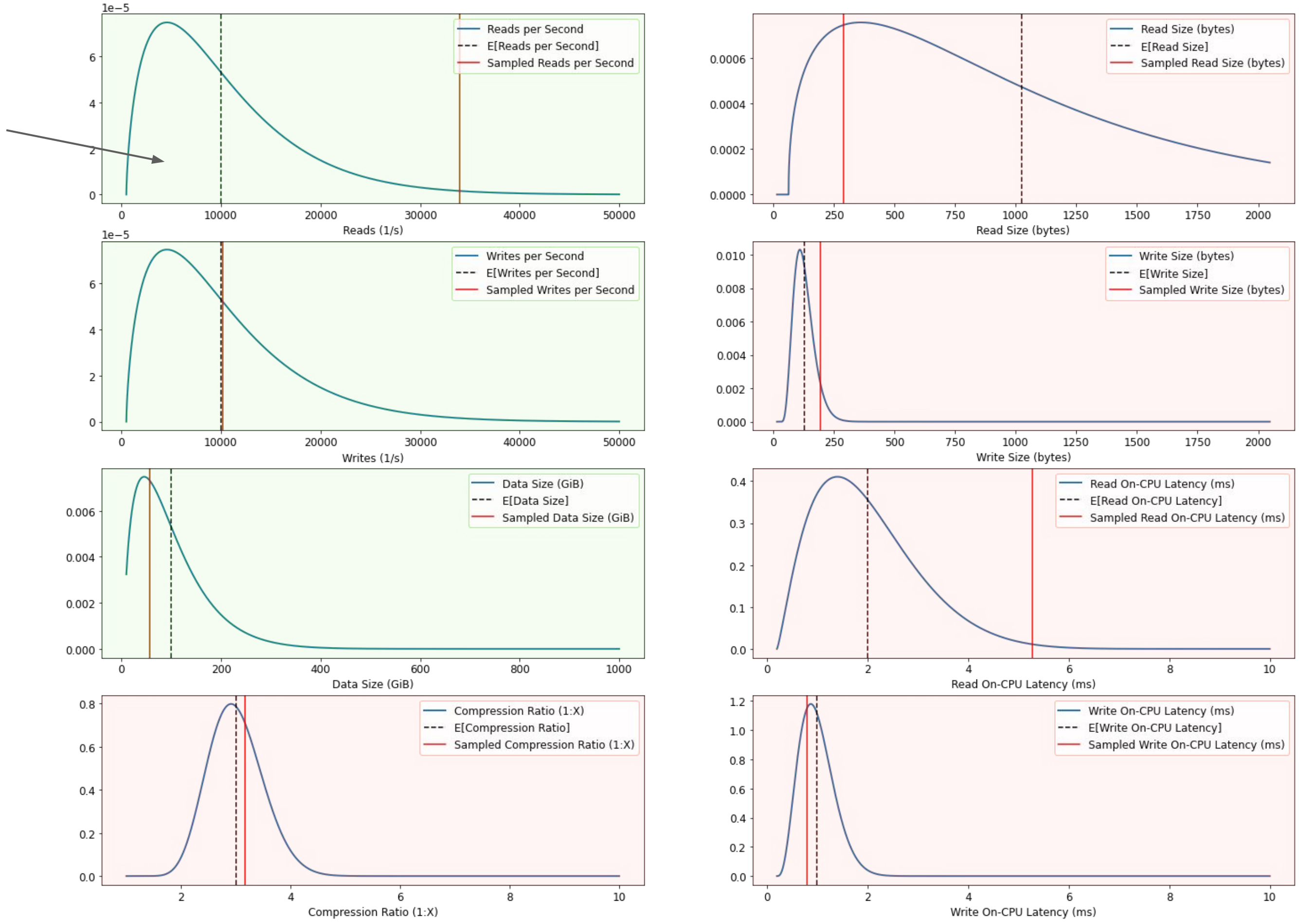
Highest Regret World

96 m5.xlarge

with 400 GiB gp2

$62,145.34

Overprovisioned!



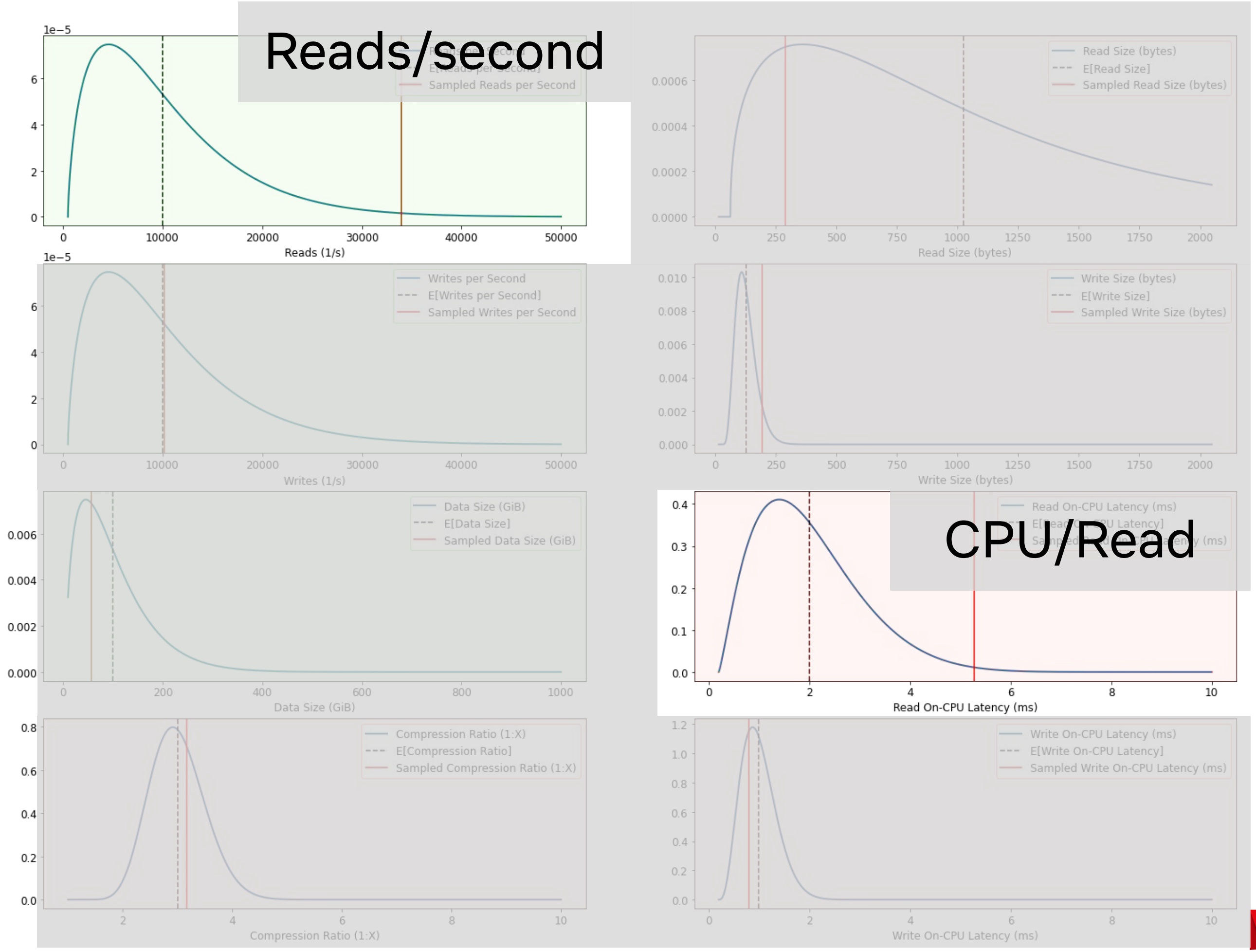
Highest Regret World

96 m5.xlarge

with 400 GiB gp2

$62,145.34

Overprovisioned!

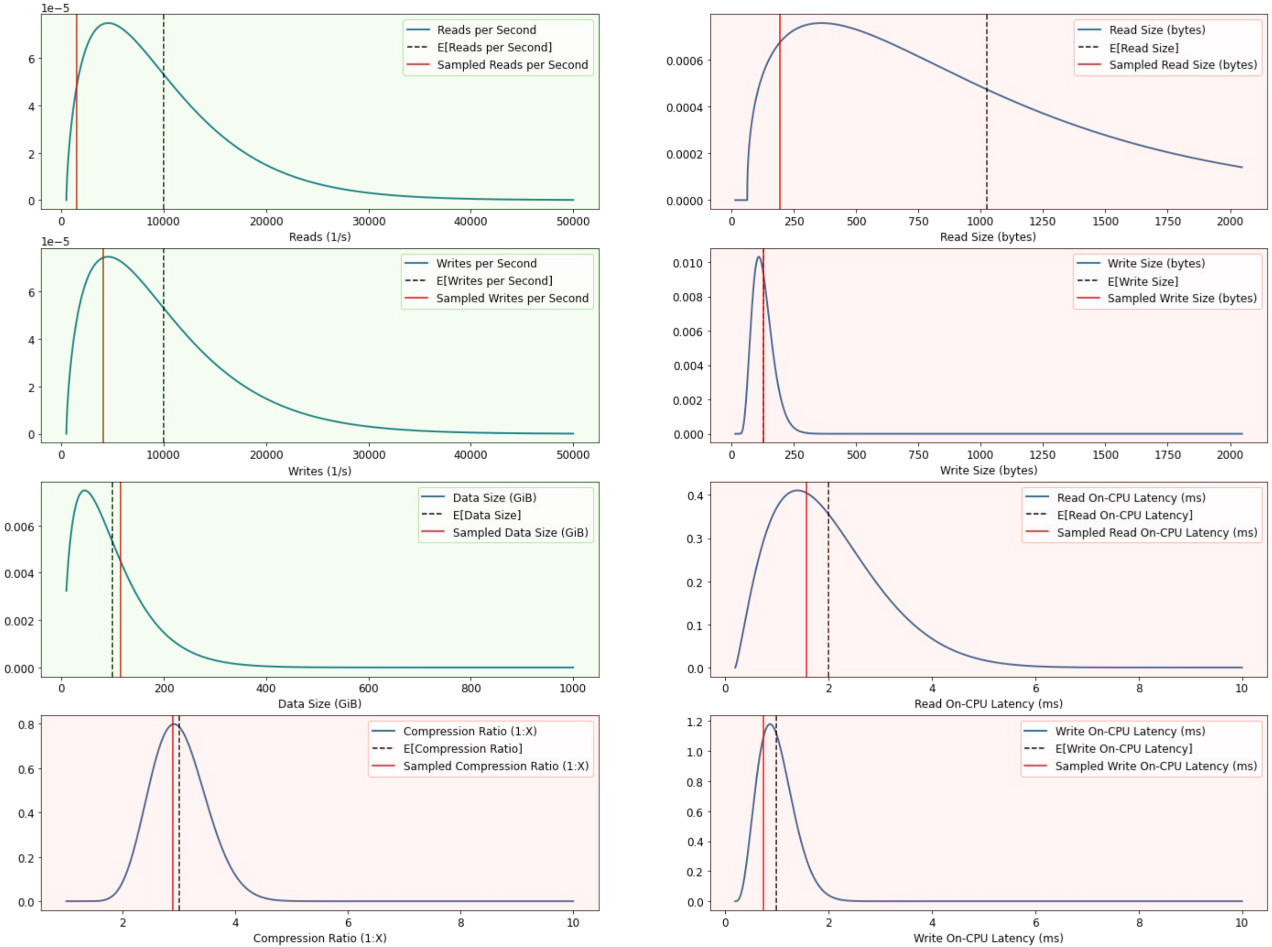


A cheap but regretful world

6 r5d.large

$2,854.34

Underprovisioned!

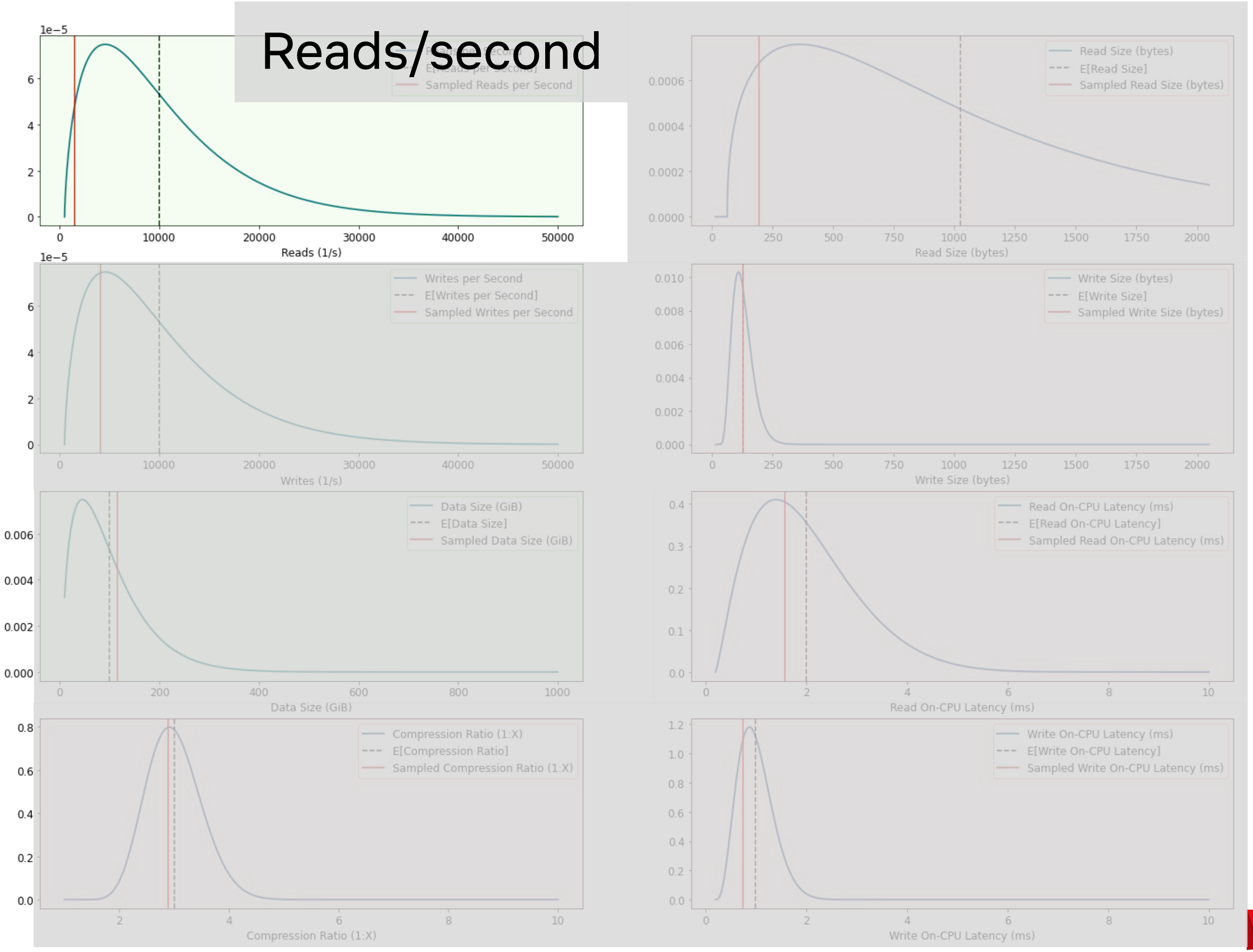


A cheap but regretful world

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

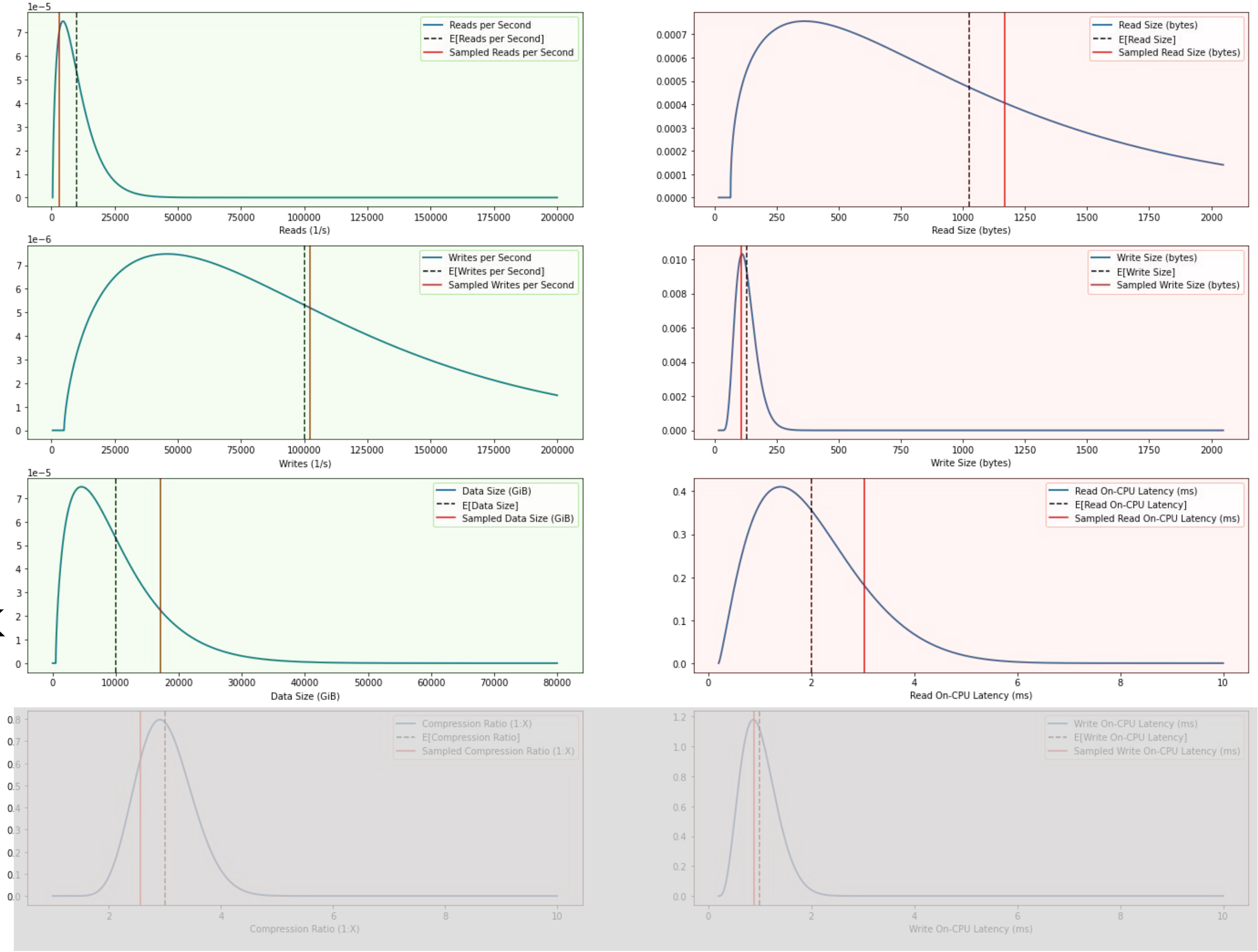
```
All Choices
-----------
{'  6 i3en.2xlarge': 13,
 '  6 i3en.3xlarge': 14,
 '  6 i3en.xlarge': 2,
 '  6 m5.8xlarge': 6,
 '  6 m5d.4xlarge': 2,
 '  6 m5d.8xlarge': 17,
 '  6 r5.4xlarge': 3,
 '  6 r5.8xlarge': 7,
 ' 12 i3en.2xlarge': 81,
 ' 12 i3en.3xlarge': 43,
 ' 12 i3en.xlarge': 17,
 ' 12 m5.4xlarge': 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': 102,
 ' 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

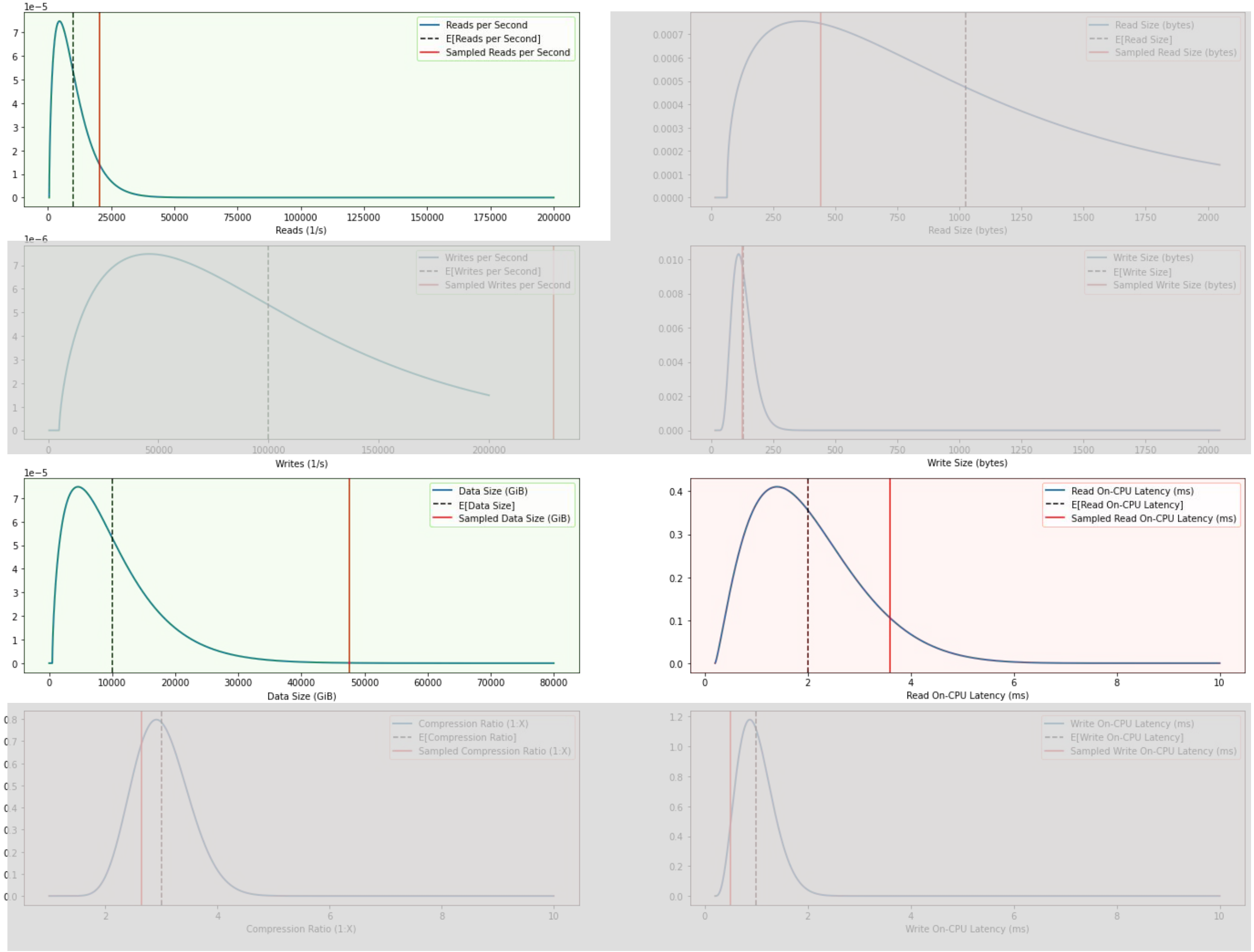


Least Regret

96 r5.8xlarge
with 1.2TiB gp2

$646,309.93

Too much money!



Monitoring

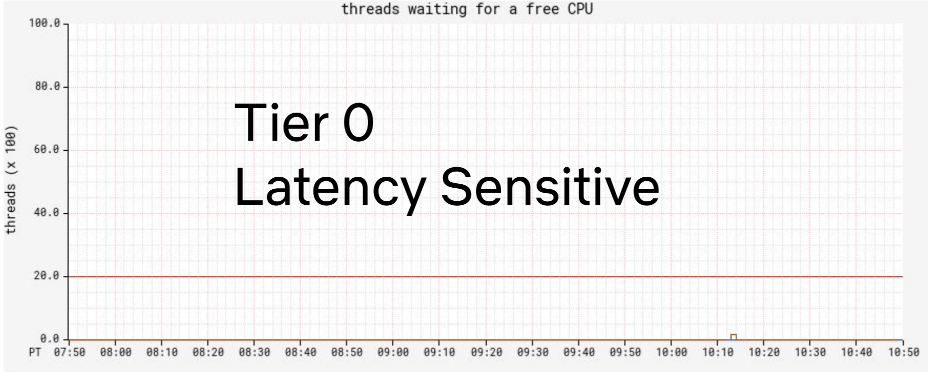
How do you know you've run out of capacity?

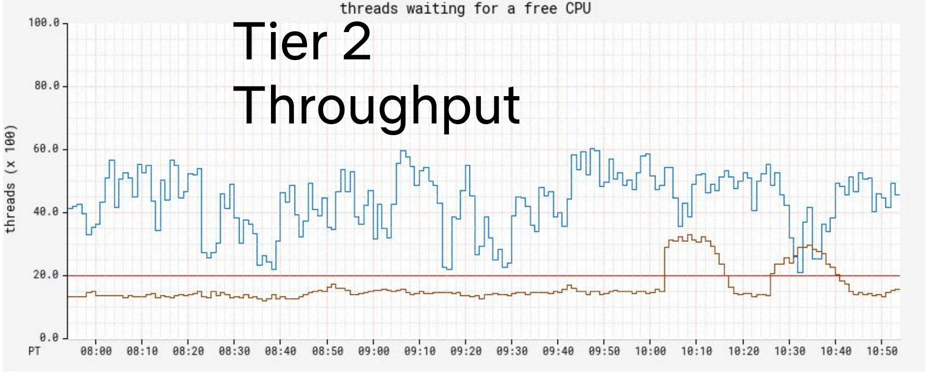
CPU

Measure
/proc/schedstat

"would additional CPUs help me"

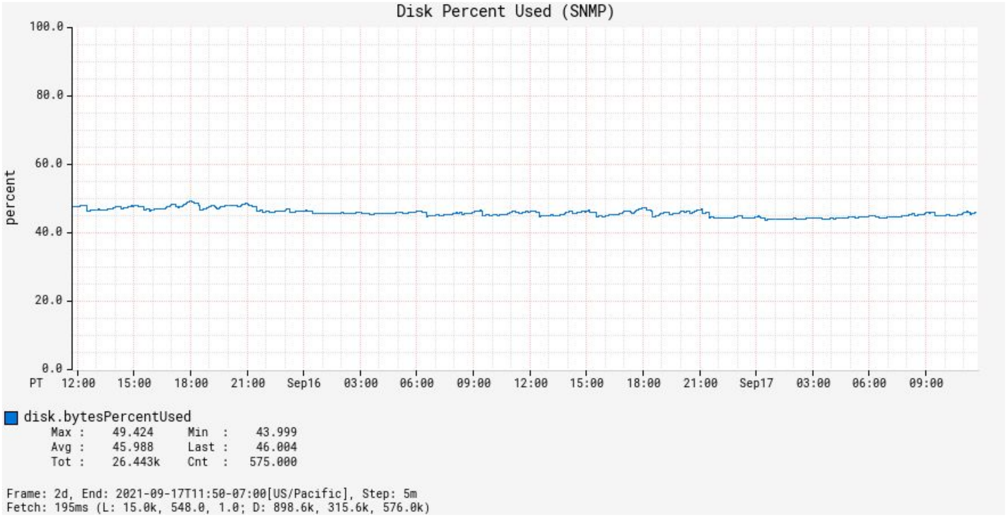
```
def gather_metric():
    # scale time spent in the scheduler by this factor
    schedstat_lines = open('/proc/schedstat').readlines()
    delays = [
      int(i.split(' ')[8]) for i
      in schedstat_lines if i.startswith('cpu')
    ]

    delays = delays or [0]
    return sum(delays) / float(len(delays))
```

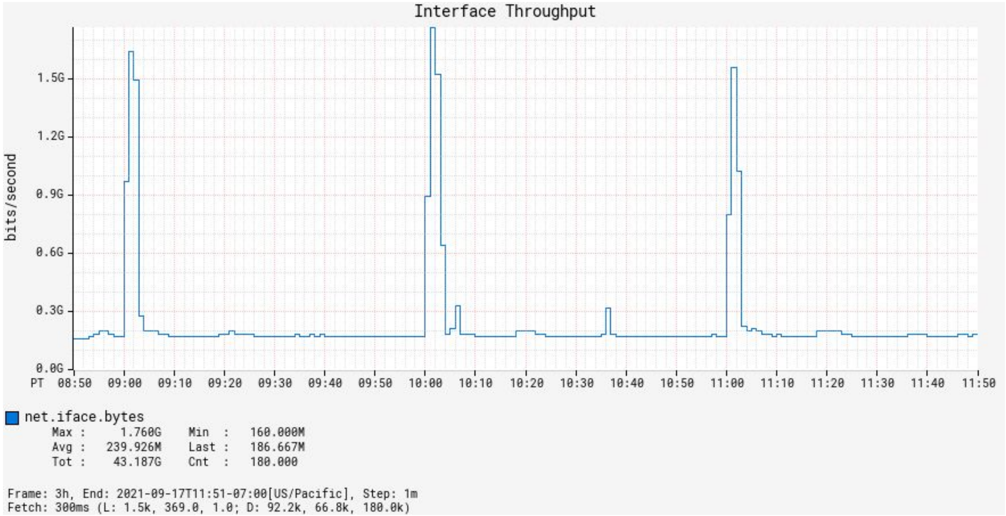




Disk
Network

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

Demo