AWS re:Invent

NFX207

Benchmarking stateful services in the cloud

Vinay Chella
Sr. Software Engineer
Netflix

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

Agenda

Introduction to Netflix stateful systems

Concept of change

Benchmarking stateful systems

Netflix benchmarking philosophies

Tools and techniques

Case studies

Speaker



Vinay Chella

Apache Cassandra Committer

Hangs out with distributed systems @ Netflix



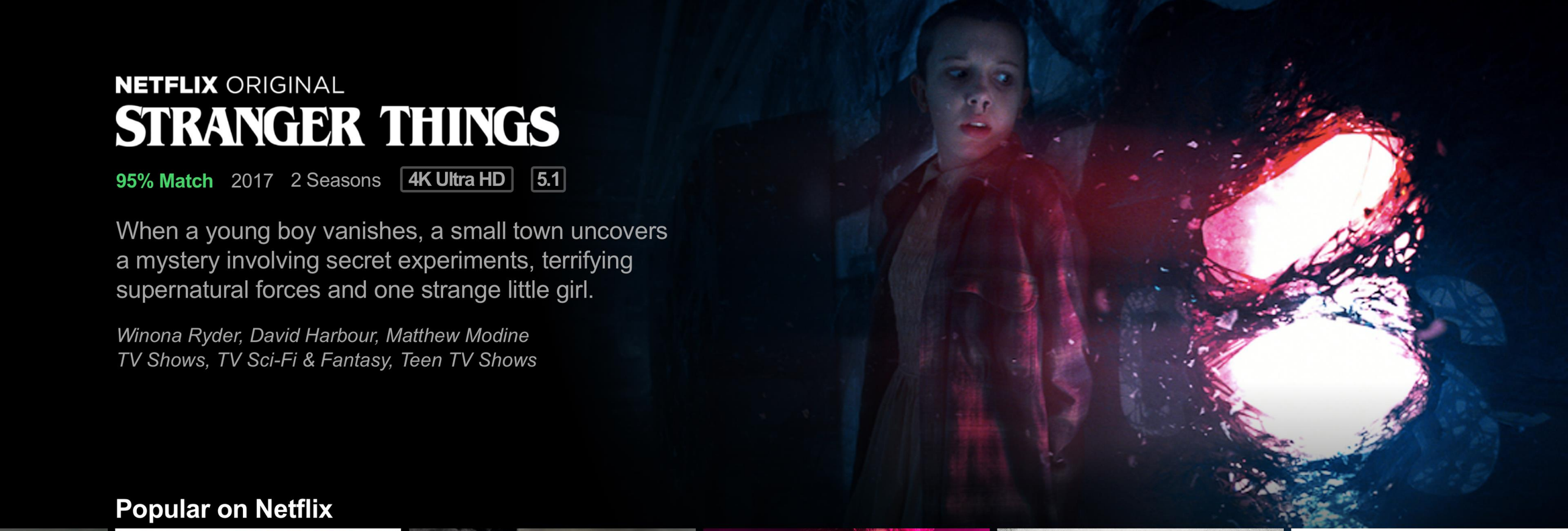






What do we do?

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.



NETFLIX ORIGINAL

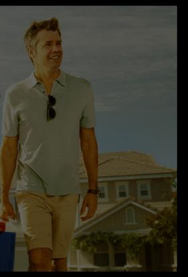
STRANGER THINGS

95% Match 2017 2 Seasons 4K Ultra HD 5.1

When a young boy vanishes, a small town uncovers a mystery involving secret experiments, terrifying supernatural forces and one strange little girl.

Winona Ryder, David Harbour, Matthew Modine
TV Shows, TV Sci-Fi & Fantasy, Teen TV Shows

Popular on Netflix












Recently Watched



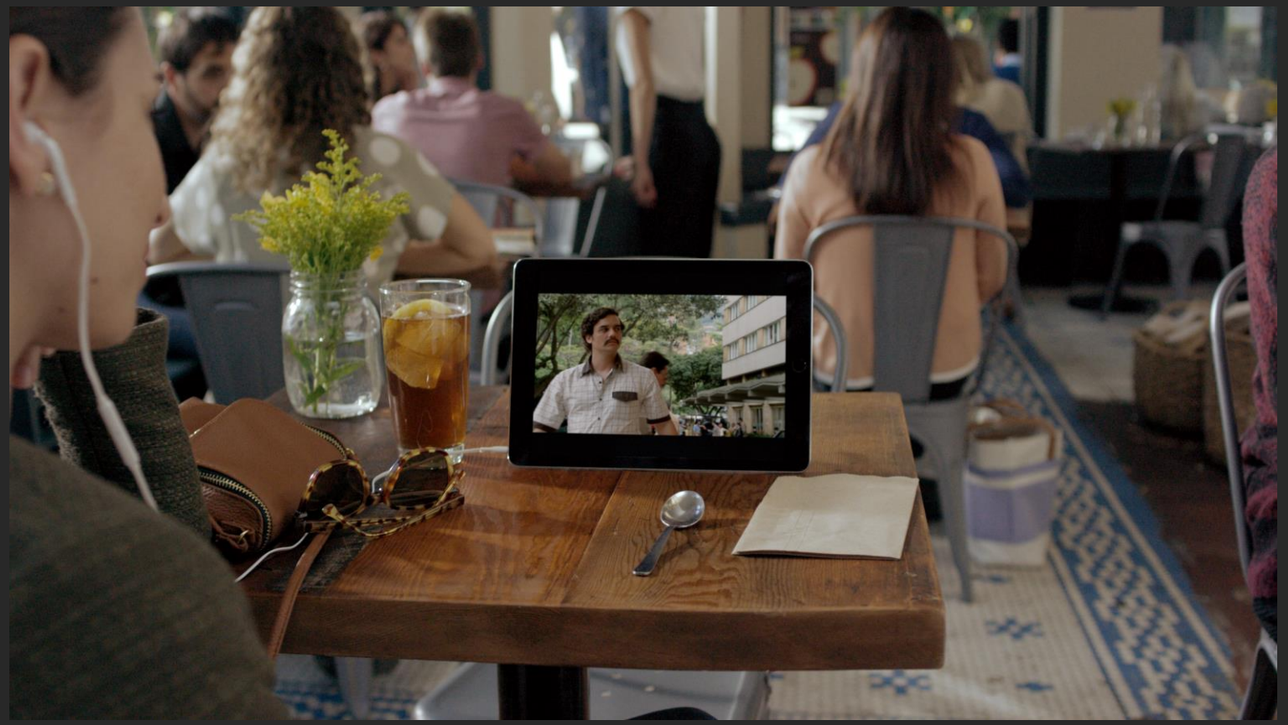











NETFLIX
NETFLIX
STRANGER THINGS
NETFLIX
NARCOS
NETFLIX
MARVEL
LUKE CAGE

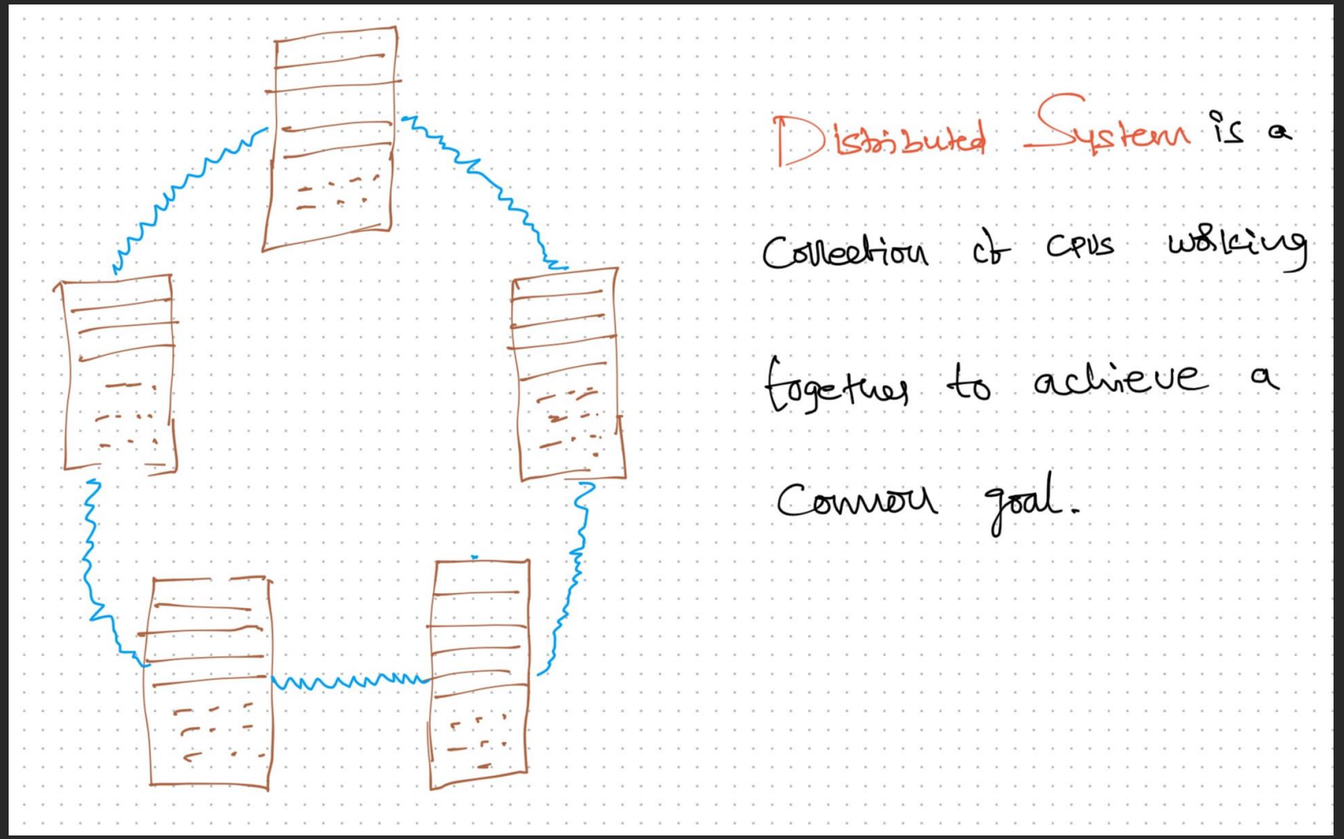
How does technology matter?



How do distributed systems help?

- Horizontally scalable
- Achieving sustainable performance while serving millions of users
- Providing seamless experience to every customer of yours
- But …

Does it really happen?



Maybe!

Maybe not!

Benchmark, to build confidence!

Stateful services @ Netflix?







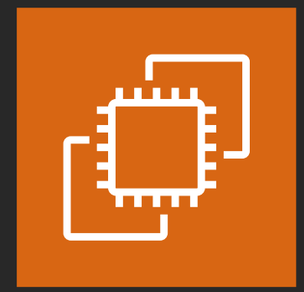
















A few more …

Scale of stateful services

Footprint in numbers

- 10s of petabytes of storage
- 100s of datastore clusters
- 10,000s of datastore nodes
- 100,000s of datastore CPU cores

Scale in numbers

- Tens of millions of operations per second
- Trillions of data records
- Non-durable stores offering sub-millisecond latencies
- Durable stores offering low-single-digit-millisecond latencies

Change

Data model

Data access pattern

New features

New datastores

Custom data stores



How do we provide sub-ms latencies in this ever-changing ecosystem?

Benchmark, to build confidence!

Why benchmark stateful services?

- Stateful services are complex mixture of several systems
- They are a hard piece of software
  - Involves state transitions
  - Different states have different kinds of impact on your resources
- Can’t be auto scaled immediately
- These systems are hard to benchmark
  - Ever-changing access patterns
  - Stateful systems depend on other characteristics like page cache, disk variability
  - Build the confidence mental model

How do stateful systems benchmark?

- How do traditional relational databases do?
  - TPC-C (The Transaction Processing Performance Council)
- How do open source distributed datastores do?
  - Apache Cassandra
  - Lucene
- What are the common tools around?
  - YCSB (Yahoo! Cloud Serving Benchmark)
  - Jepsen

TPC-C (The Transaction Processing Performance Council)

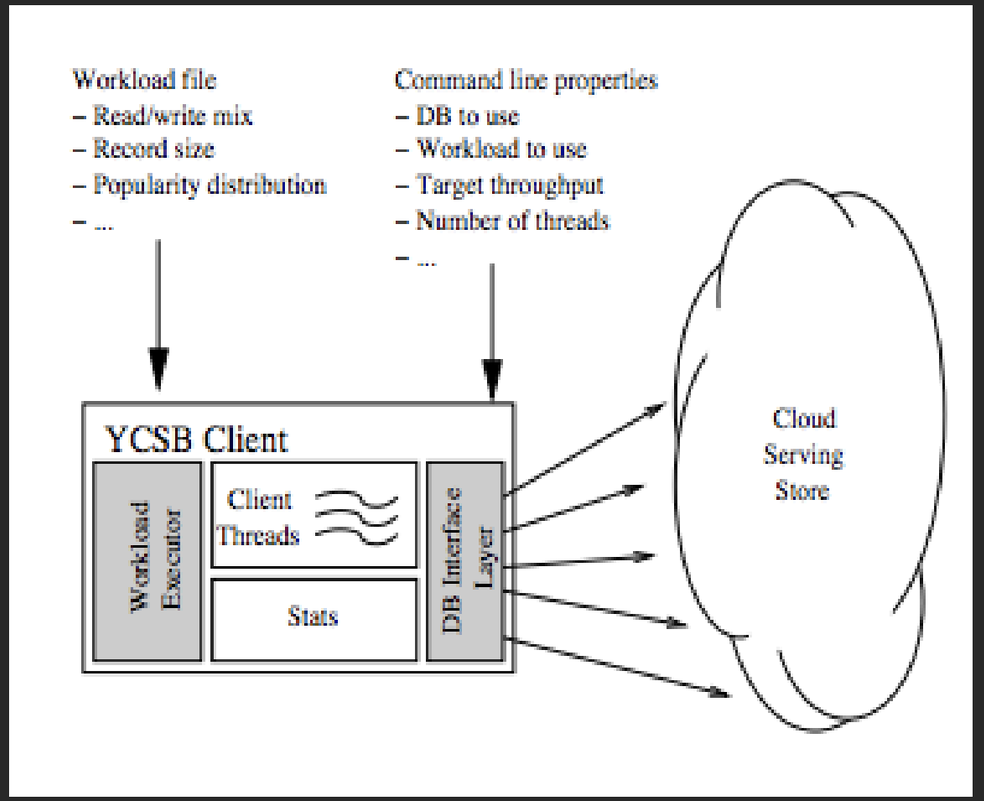
- Standard OLTP benchmark
- Centered around database transactions
- Focuses on
  - Multiple types of transactions of varying complexity
  - Complex database structures
  - Higher levels of contention for data access and update
  - Non-uniform patterns of data access to simulate data hot spots
  - Data index-ability
  - Data presentation capabilities

Apache Cassandra benchmarking

- Cassandra-stress
  - Stress testing tool for basic benchmarking and load testing Apache Cassandra
- Replay testing
  - A black box technique for testing database functionality, a simple side-by-side comparison of a workload using two versions of the same database
- Harry
  - A fuzz testing tool for Apache Cassandra
- Property-based testing
  - Property-based testing combines fuzz testing and assertions to explore a state space using randomly generated input

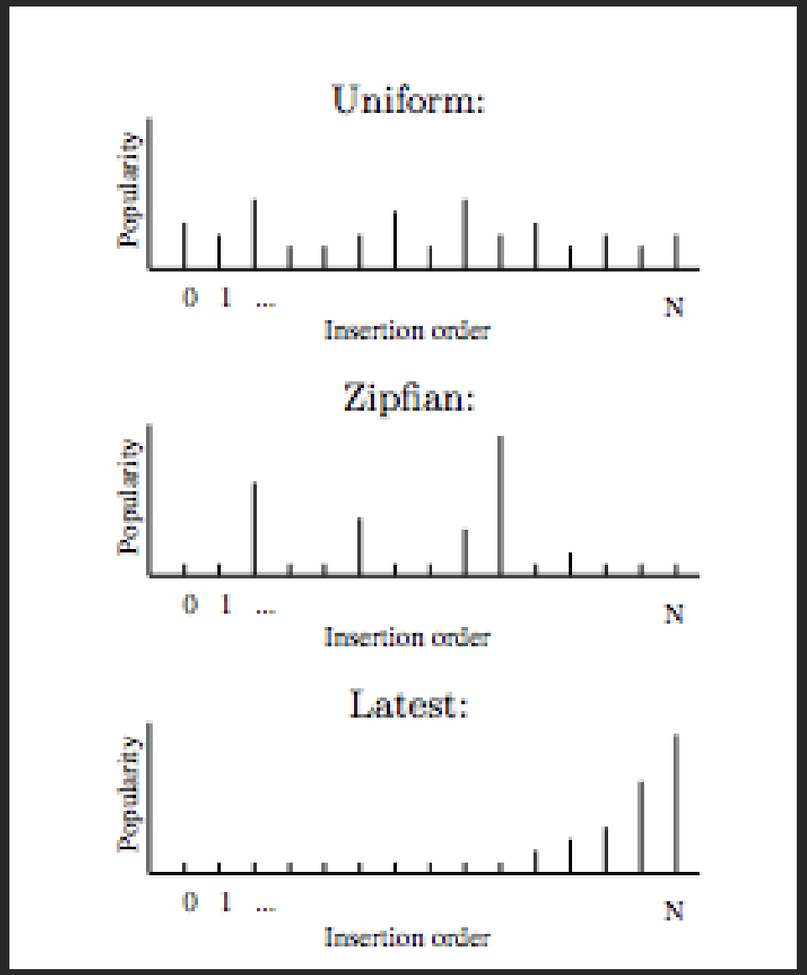
YCSB (Yahoo! Cloud Serving Benchmark)

- A standard benchmarking framework to evaluate the performance of stateful services in cloud
- Two major components:
  - The YCSB client
  - The core workloads



YCSB (Yahoo! Cloud Serving Benchmark)

- Workload generator supports various distribution



Jepsen

- Tool by Kyle Kingsbury. A framework for distributed systems verification, with fault injection
- Benchmarks partition tolerance of your systems
- Testing under distributed systems failure modes (faulty networks, unsynchronized clocks, and partial failure)

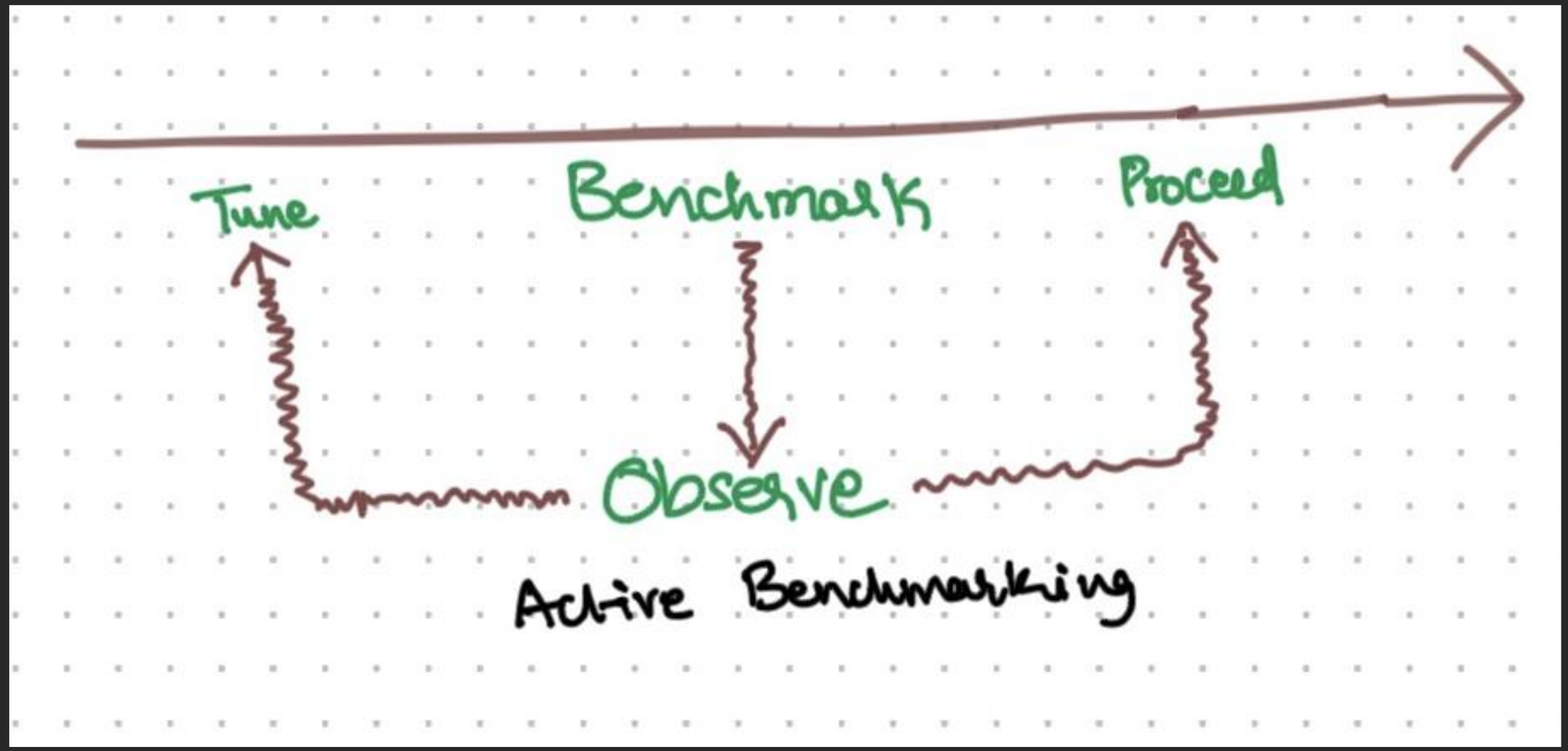
How do we approach benchmarking?

Benchmarking models

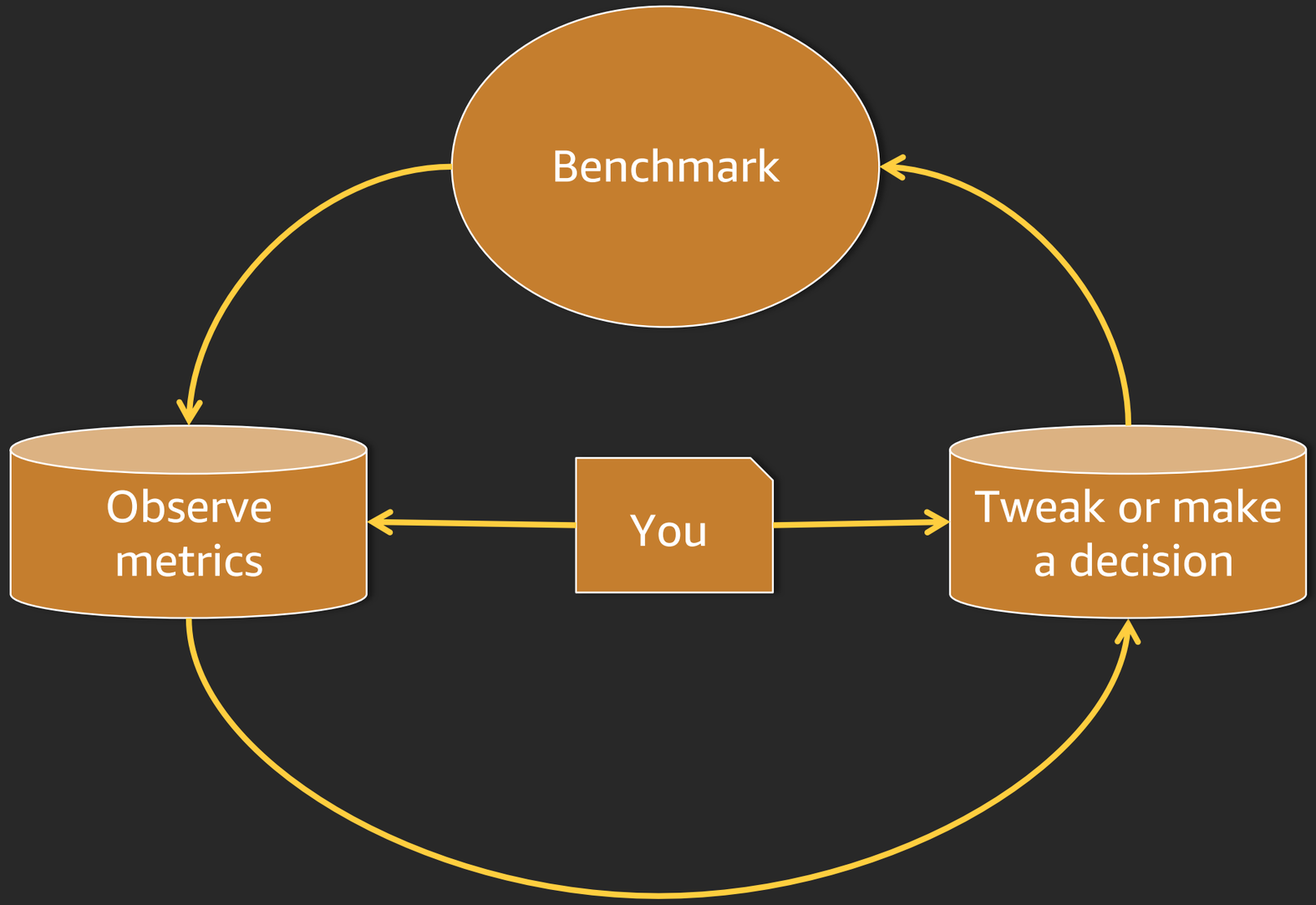
Active Benchmarking

Passive Benchmarking

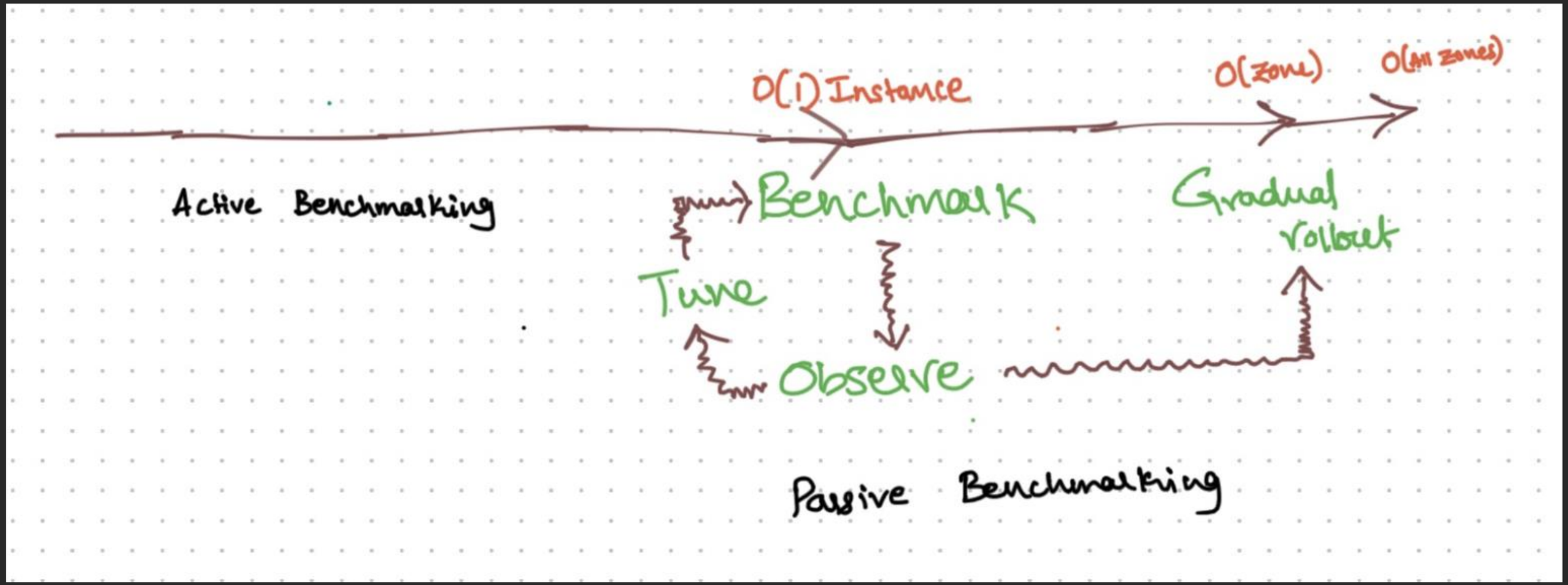
Benchmarking – Active



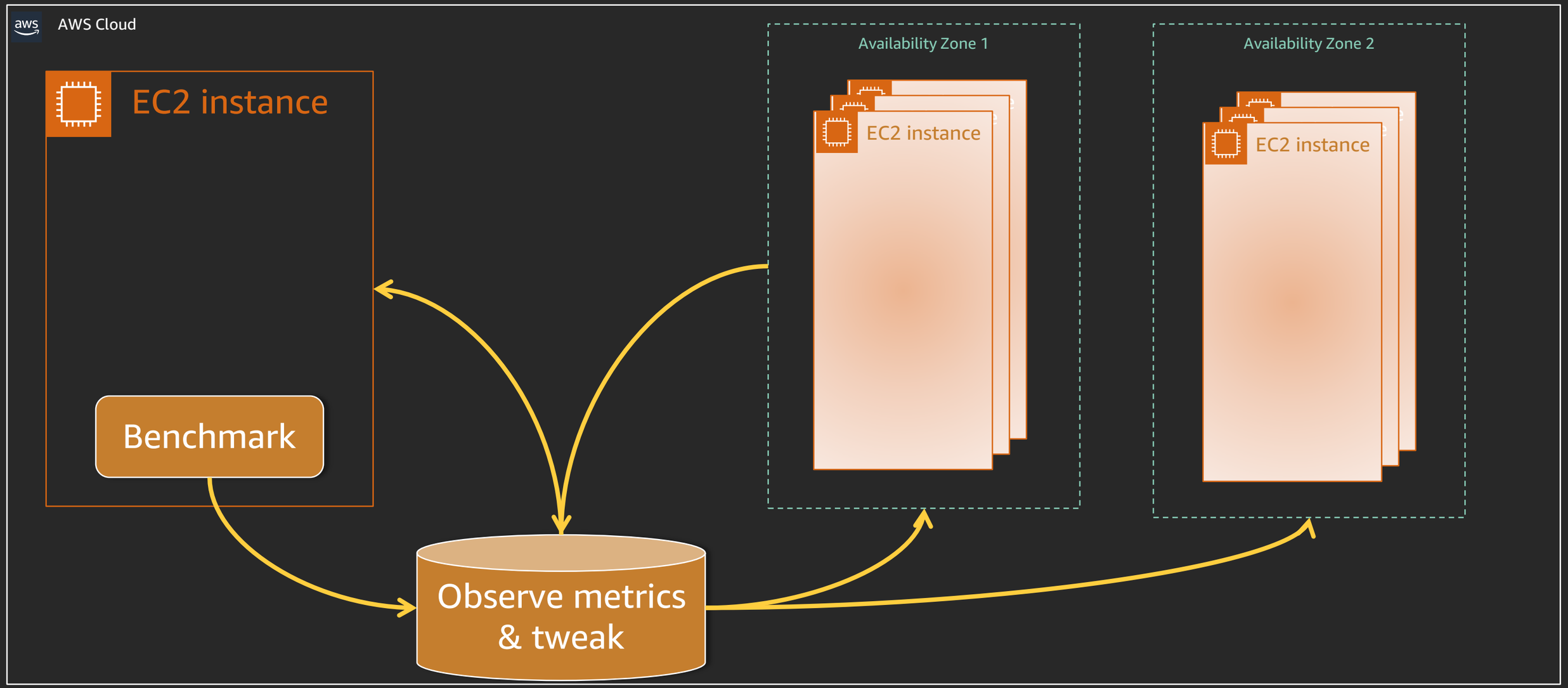
Active benchmarking: Mental model



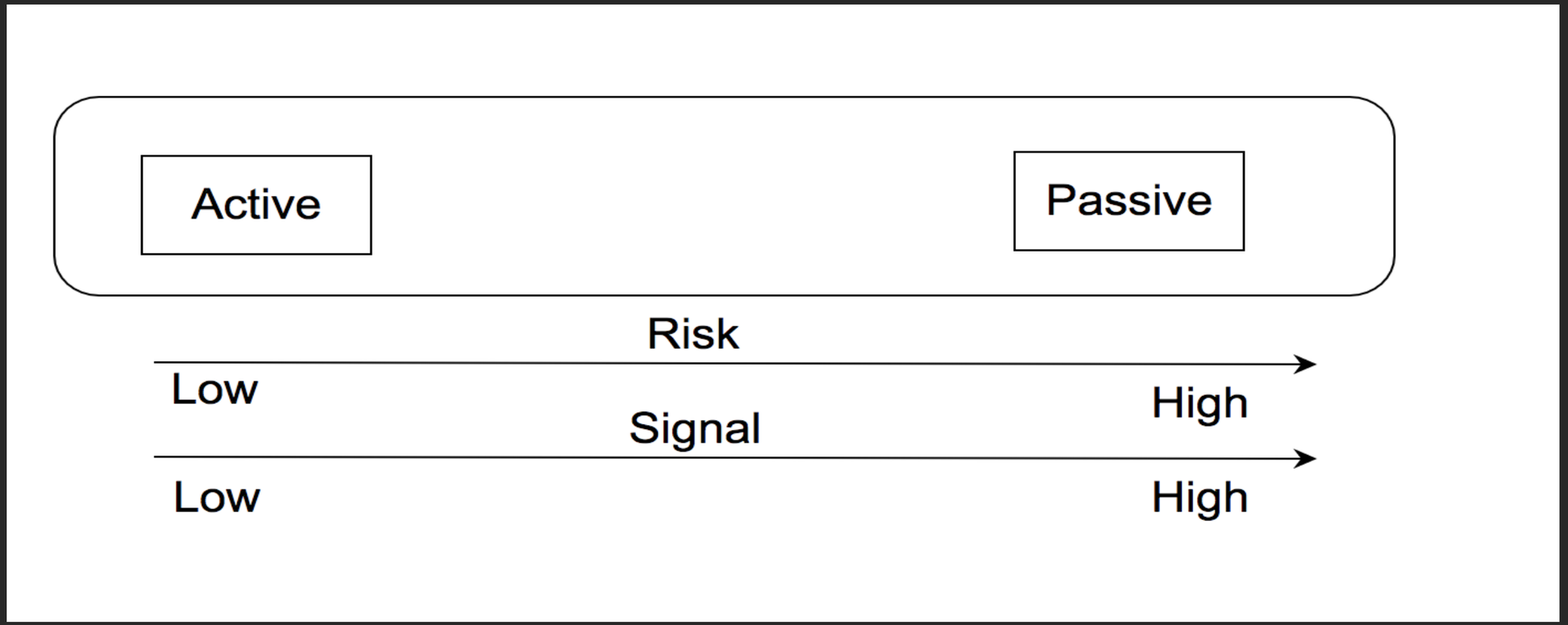
Benchmarking – Passive



Passive benchmarking: Mental model



Tradeoffs



Summary: Benchmarking models

Active benchmarking

- Generate load to benchmark
  - Generative load
  - Random load
- Observe the metrics
- Identify bottlenecks
- Tune your services

Passive benchmarking

- Production load is the true load
- Deploy on one instance
- Observe the metrics
- Identify bottlenecks
- Buy what is missing
  - CPU, Network, Memory
- Incremental rollouts

Netflix benchmark philosophy



Automate



Continuous integration



Every stage of product lifecycle



Not a discrete action, it is a continuous process

What tools do we have to achieve this?

Active benchmarking

- NDBench
  - Benchmarking for new data stores/use cases
  - CI benchmarking
- Preflight check
  - Benchmarking in production before launching new services
- One-off benchmarking tools
- CI benchmarking

Passive benchmarking

- AWS bad IO detector
  - Benchmarking all the time in production
- jvmquake
  - Continuously benchmarking the performance of your JVM in production systems
- Declarative control planes

Netflix benchmark philosophy – Tools

Active benchmarking

- **NDBench**
- CI Benchmarking
- Preflight check

Passive benchmarking

- AWS bad IO detector
- jvmquake
- Declarative control planes

Active benchmarking

NDBench

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

What is NDBench?

Netflix Data Benchmark (NDBench) is a **pluggable** **cloud-enabled** benchmarking tool that can be used across **any** **stateful service**



NDBench



- Active benchmarking tool
- Supports different data stores via plugins (available plugins)
- Dynamically tunable RPS and configuration
- Pluggable load patterns
  - random, sliding window, zipfian

https://github.com/netflix/ndbench

Why NDBench?

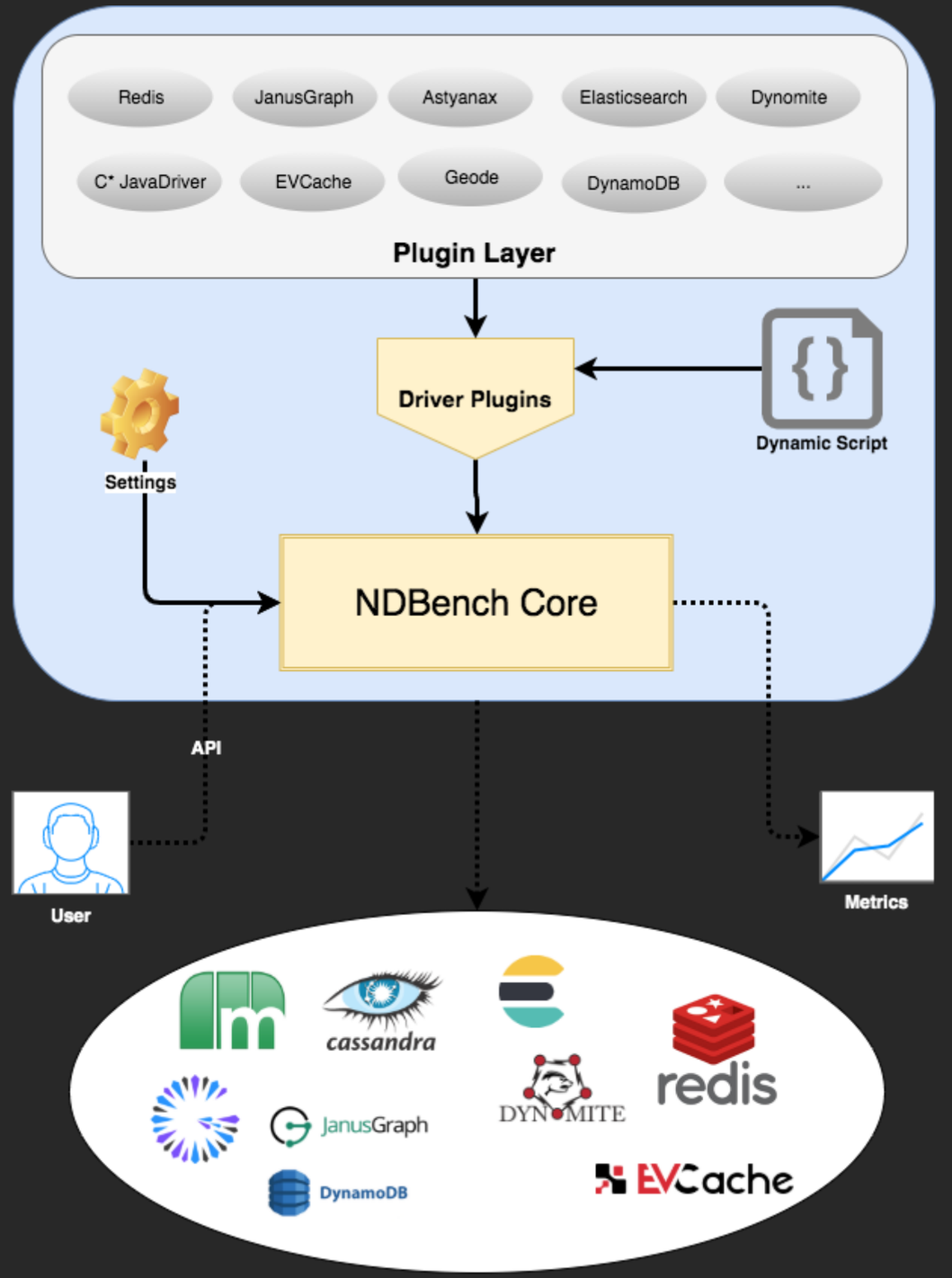
- Runs for infinite amount of time
    - Not discrete action and it is continuous action
- Dynamically tunable configurations
- Pluggable patterns & loads
- Pluggable architecture enables working with various datastores (Cassandra, Dynomite, ElasticSearch etc.)
- Auto tuning and Auto detecting data models

Architecture

Plugin: Client drivers

Core: Workload generator

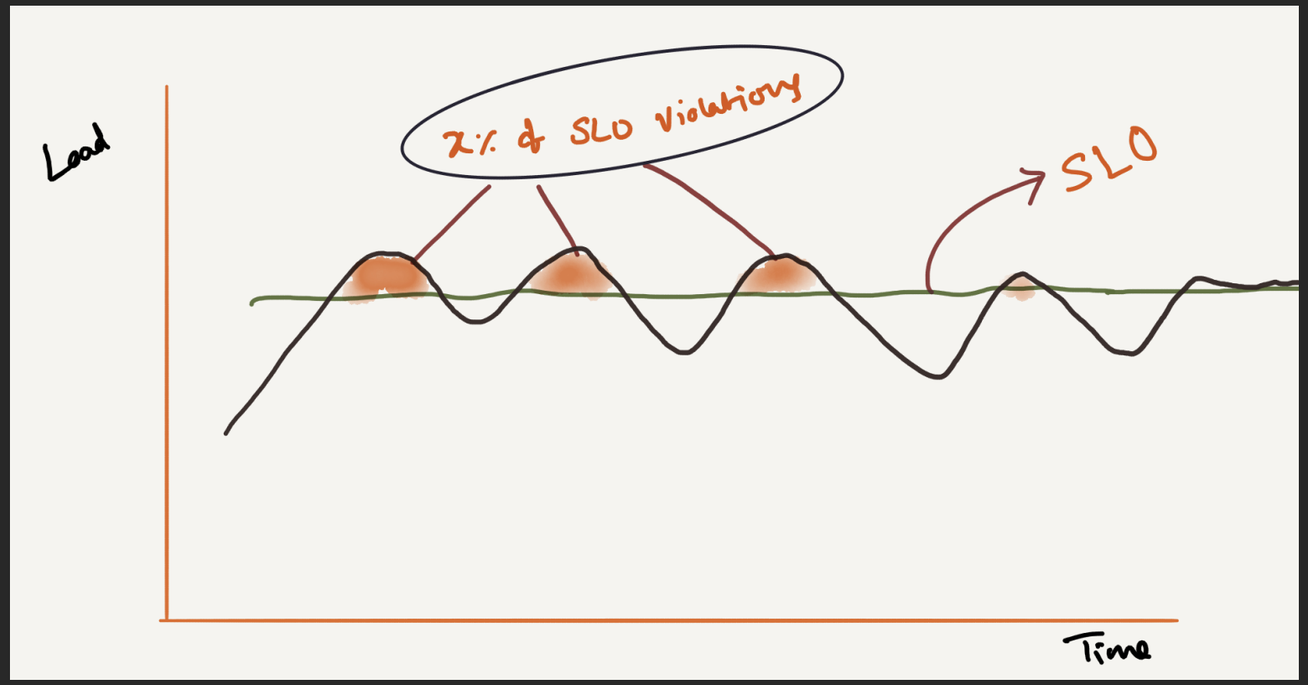
Metrics: Pluggable metrics layer



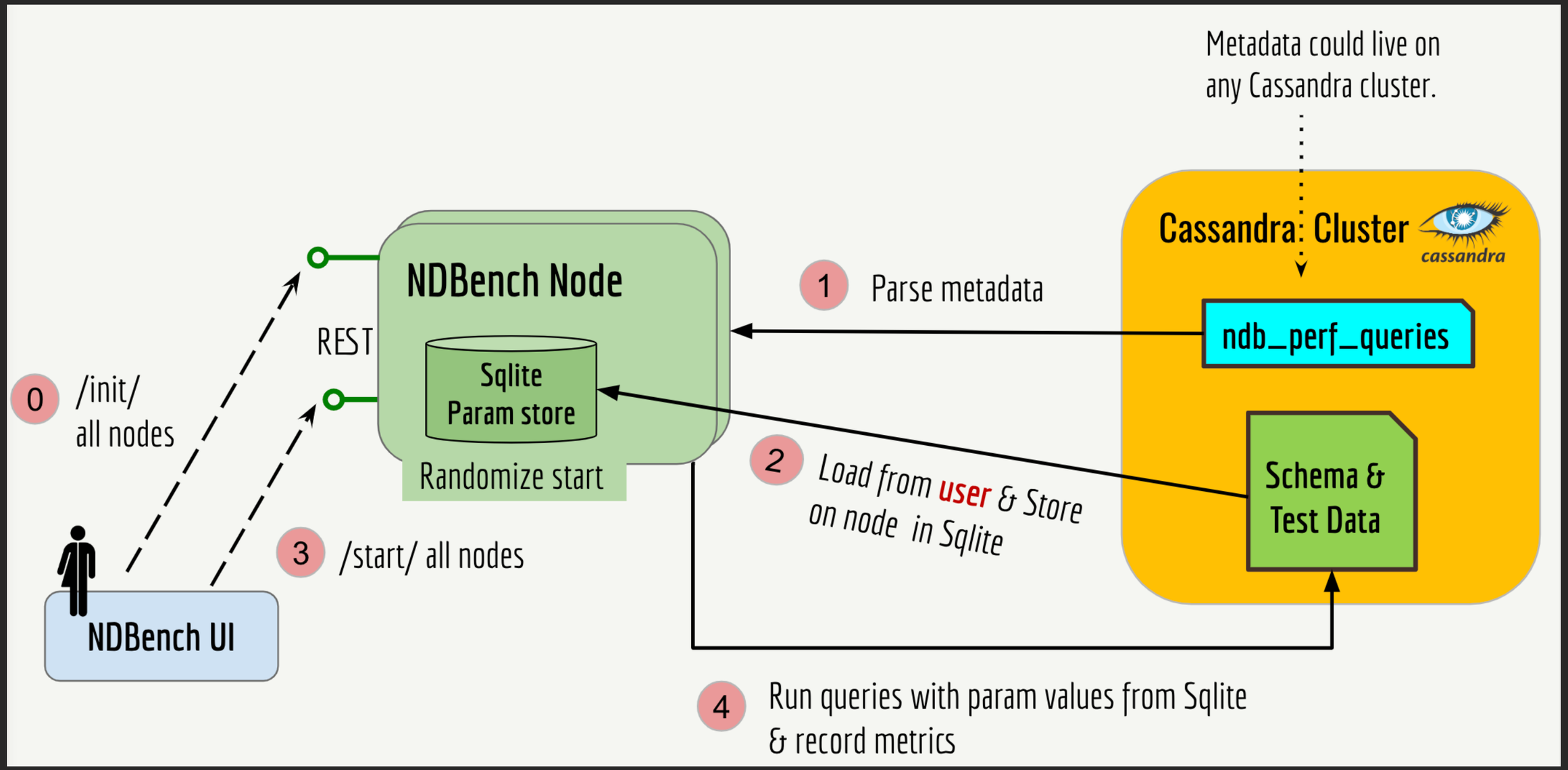
https://github.com/netflix/ndbench

Auto tuning

- Find the appropriate capacity that the service can sustain
- E.g., rps/wps, number of documents indexed/sec
- After a set % of SLA violations, NDBench stops stepping up the load with exponential backoff



Auto-detecting data models



NDBench – Ease of use

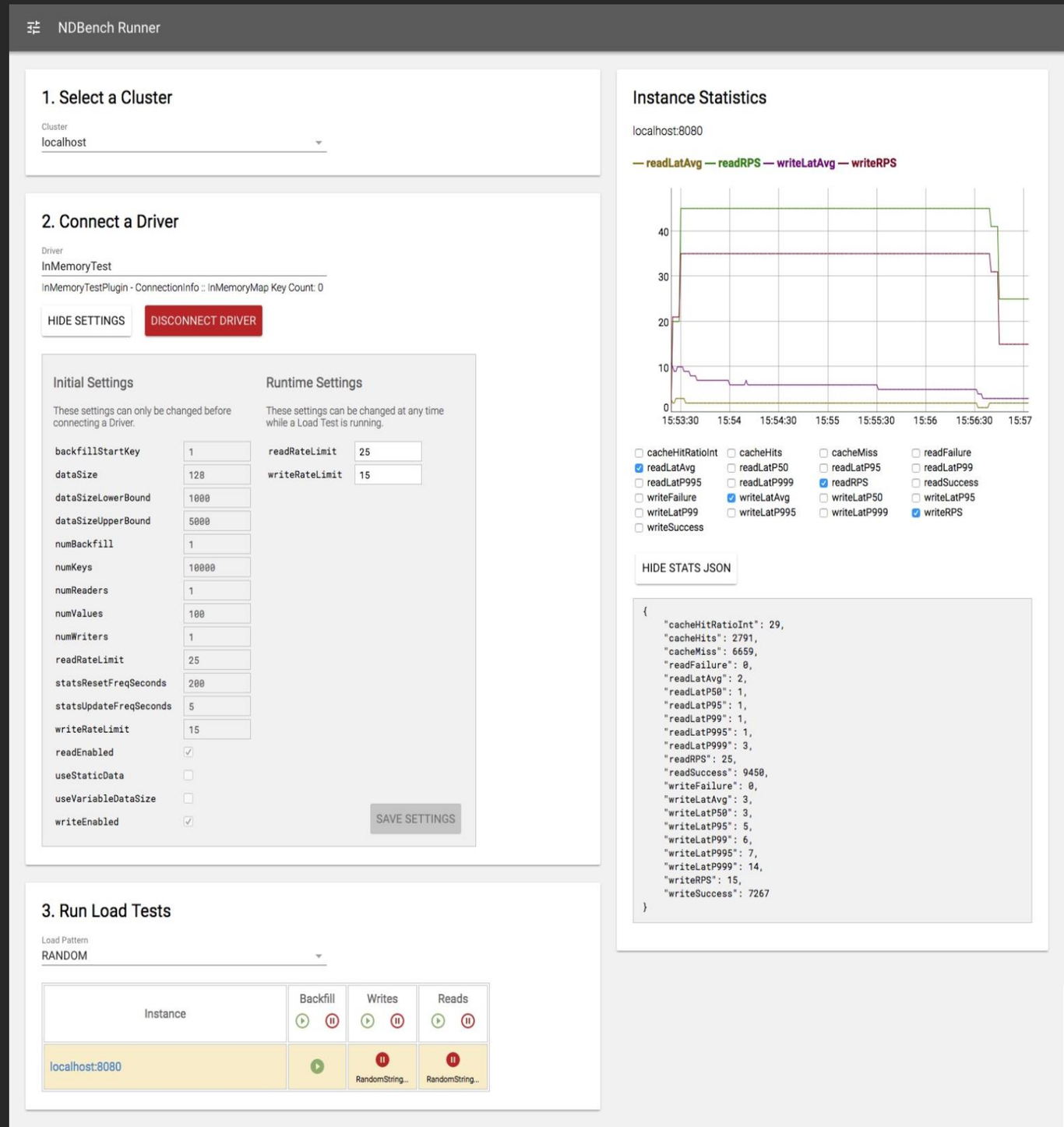
Configuring an NDBench workload

As easy as launching a pre-baked AMI



Using NDBench

via UI portal



NDBench Runner

test.us-east-1

selecting a cluster

1. Select a Cluster

Cluster

NDB_QCON

2. Connect a Driver

Driver

A6XGeneric

Cluster Name - cass_qcon1 : Keyspace Name - pappyperftest : CF Name - test1 ::: ColsPerRow - 10 : ReadCL - CL_LOCAL_ONE : WriteCL - CL_LOCAL_ONE

SHOW SETTINGS | DISCONNECT DRIVER

3. Run Load Tests

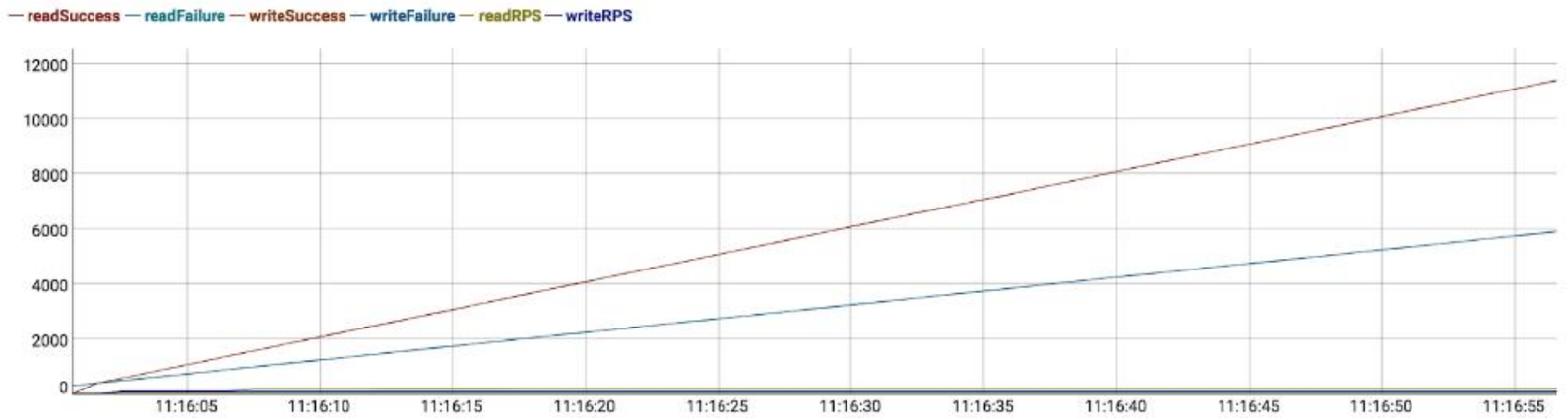
Load Pattern

RANDOM

Instance	Backfill	Writes	Reads
[redacted].compute-1.amazonaws.com:7001	▶	⏸ RandomStri...	⏸ RandomStri...
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶

Instance Statistics

ec2-34-203-200-0.compute-1.amazonaws.com:7001



☑ readSuccess ☑ readFailure ☑ writeSuccess ☑ writeFailure ☐ cacheHits ☐ cacheMiss ☑ readRPS ☑ writeRPS ☐ stats ☐ readLatAvg
☐ readLatP50 ☐ readLatP95 ☐ readLatP99 ☐ readLatP995 ☐ readLatP999 ☐ writeLatAvg ☐ writeLatP50 ☐ writeLatP95 ☐ writeLatP99 ☐ writeLatP995
☐ writeLatP999 ☐ cacheHitRatioInt ☐ documentation ☐ status

SHOW STATS JSON

NDBench Runner

test.us-east-1

1. Select a Cluster

Cluster

NDB_QCON

2. Connect a Driver

Driver

A6XGeneric

Cluster Name - cass_qcon1 : Keyspace Name - pappyperftest : CF Name - test1 ::: ColsPerRow - 10 : ReadCL - CL_LOCAL_ONE : WriteCL - CL_LOCAL_ONE

SHOW SETTINGS | DISCONNECT DRIVER

3. Run Load Tests

Load Pattern

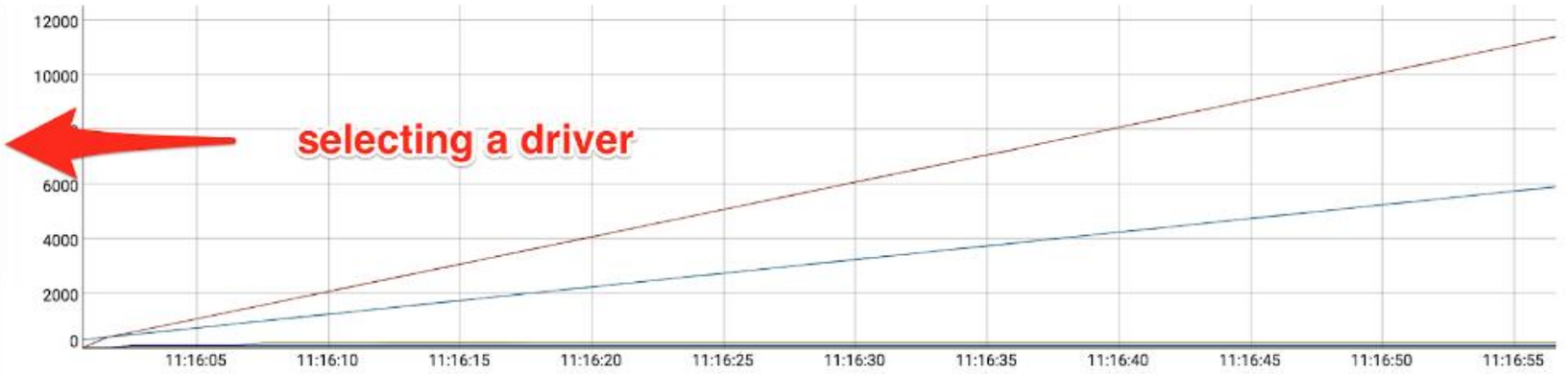
RANDOM

Instance	Backfill	Writes	Reads
[redacted].compute-1.amazonaws.com:7001		RandomStri...	RandomStri...
ec2-[redacted]compute-1.amazonaws.com:7001			
ec2-[redacted].compute-1.amazonaws.com:7001			
ec2-[redacted]compute-1.amazonaws.com:7001			
ec2-[redacted]compute-1.amazonaws.com:7001			
ec2-[redacted]compute-1.amazonaws.com:7001			
ec2-[redacted]compute-1.amazonaws.com:7001			
ec2-[redacted].compute-1.amazonaws.com:7001			

Instance Statistics

ec2-34-203-200-0.compute-1.amazonaws.com:7001

readSuccess readFailure writeSuccess writeFailure readRPS writeRPS



selecting a driver

readSuccess readFailure writeSuccess writeFailure cacheHits cacheMiss readRPS writeRPS stats readLatAvg
readLatP50 readLatP95 readLatP99 readLatP995 readLatP999 writeLatAvg writeLatP50 writeLatP95 writeLatP99 writeLatP995
writeLatP999 cacheHitRatioInt documentation status

SHOW STATS JSON

2. Connect a Driver

Driver

- DynoPlugin
- ES
- ElassandraCassJavaDriverPlugin
- InMemoryTest
- MaterializedView
- Custom Script

Backfill Writes Reads

NDBench Runner

test.us-east-1

1. Select a Cluster

Cluster

NDB_QCON

2. Connect a Driver

Driver

A6XGeneric

Cluster Name - cass_qcon1 : Keyspace Name - pappyperftest : CF Name - test1 ::: ColsPerRow - 10 : ReadCL - CL_LOCAL_ONE : WriteCL - CL_LOCAL_ONE

SHOW SETTINGS | DISCONNECT DRIVER

3. Run Load Tests

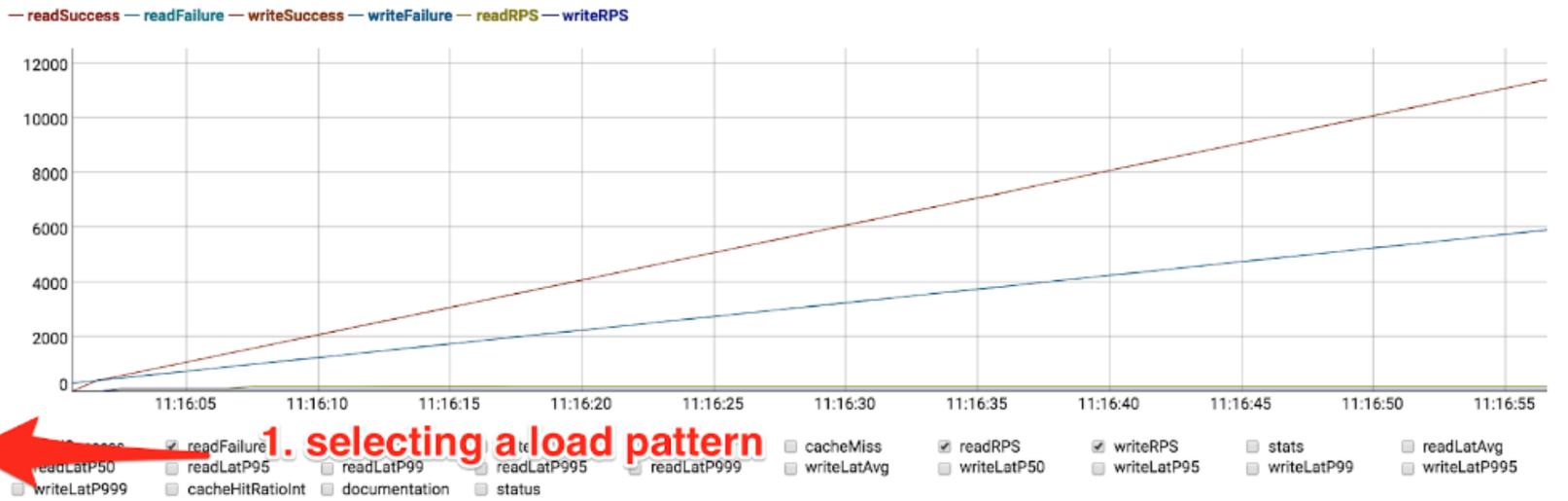
Load Pattern

RANDOM

Instance	Backfill	Writes	Reads
.compute-1.amazonaws.com:7001	▶	⏸ RandomStri...	⏸ RandomStri...
ec2-[redacted]compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted]compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted]compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted]compute-1.amazonaws.com:7001	▶	▶	▶
ec2[redacted]compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶

Instance Statistics

ec2-34-203-200-0.compute-1.amazonaws.com:7001



readFailure cacheMiss readRPS writeRPS stats readLatAvg
readLatP50 readLatP95 readLatP99 readLatP995 readLatP999 writeLatAvg writeLatP50 writeLatP95 writeLatP99 writeLatP995
writeLatP999 cacheHitRatioInt documentation status

SHOW STATS JSON

1. selecting a load pattern

2.starting the workload

NDBench Runner

test.us-east-1

1. Select a Cluster

Cluster

NDB_QCON

Monitor

2. Connect a Driver

Driver

A6XGeneric

Cluster Name - cass_qcon1 : Keyspace Name - pappyperftest : CF Name - test1 ::: ColsPerRow - 10 : ReadCL - CL_LOCAL_ONE : WriteCL - CL_LOCAL_ONE

SHOW SETTINGS | DISCONNECT DRIVER

3. Run Load Tests

Load Pattern

RANDOM

Instance	Backfill	Writes	Reads
[redacted].compute-1.amazonaws.com:7001	▶	⏸ RandomStri...	⏸ RandomStri...
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶
ec2-[redacted].compute-1.amazonaws.com:7001	▶	▶	▶

Instance Statistics

ec2-34-203-200-0.compute-1.amazonaws.com:7001

— readSuccess — readFailure — writeSuccess — writeFailure — readRPS — writeRPS



☑ readSuccess ☑ readFailure ☑ writeSuccess ☑ writeFailure ☐ cacheHits ☐ cacheMiss ☑ readRPS ☑ writeRPS ☐ stats ☐ readLatAvg
☐ readLatP50 ☐ readLatP95 ☐ readLatP99 ☐ readLatP995 ☐ readLatP999 ☐ writeLatAvg ☐ writeLatP50 ☐ writeLatP95 ☐ writeLatP99 ☐ writeLatP995
☐ writeLatP999 ☐ cacheHitRatioInt ☐ documentation ☐ status

SHOW STATS JSON

Using NDBench

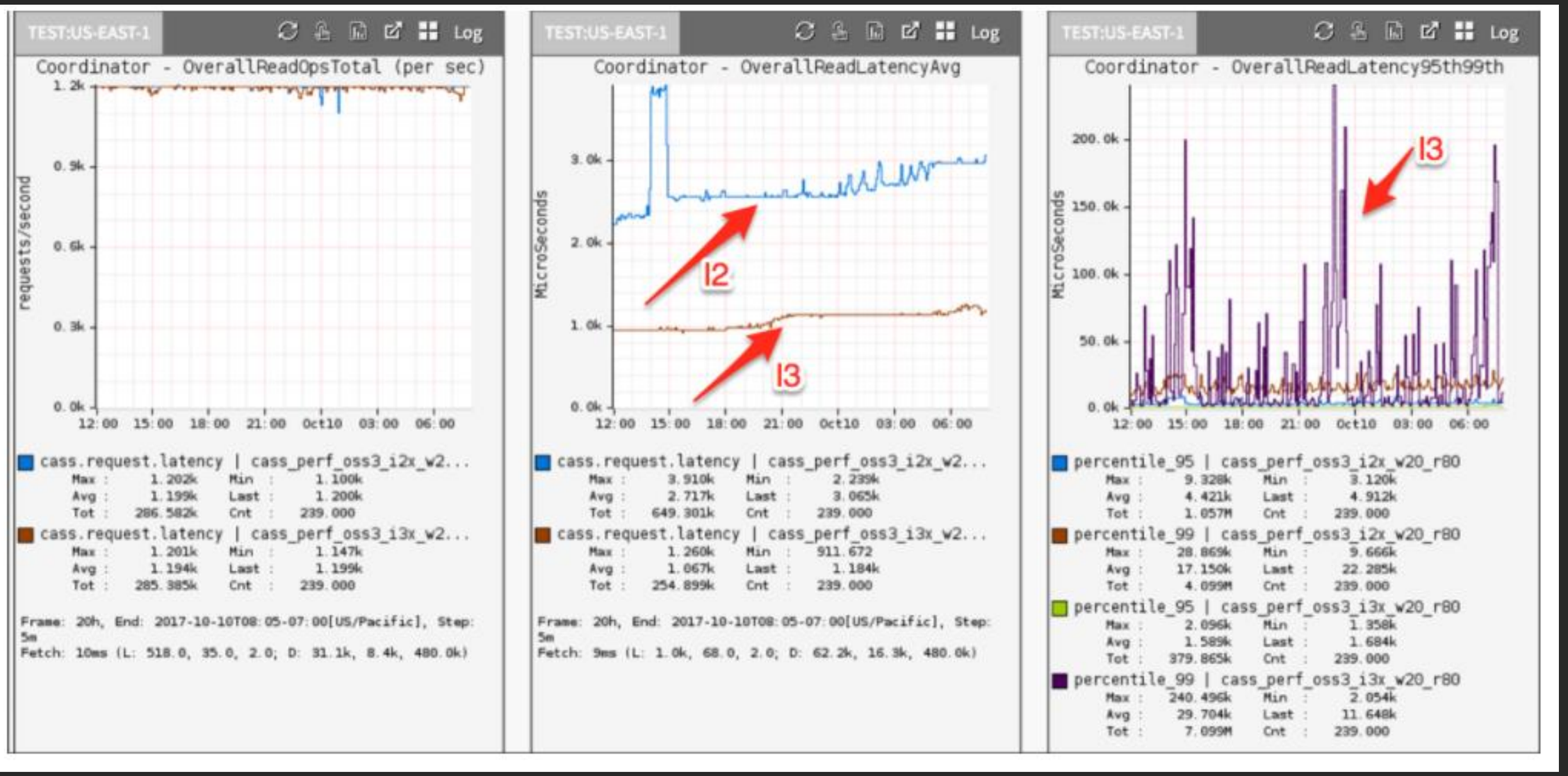
via API

NdBench Driver Resource REST API

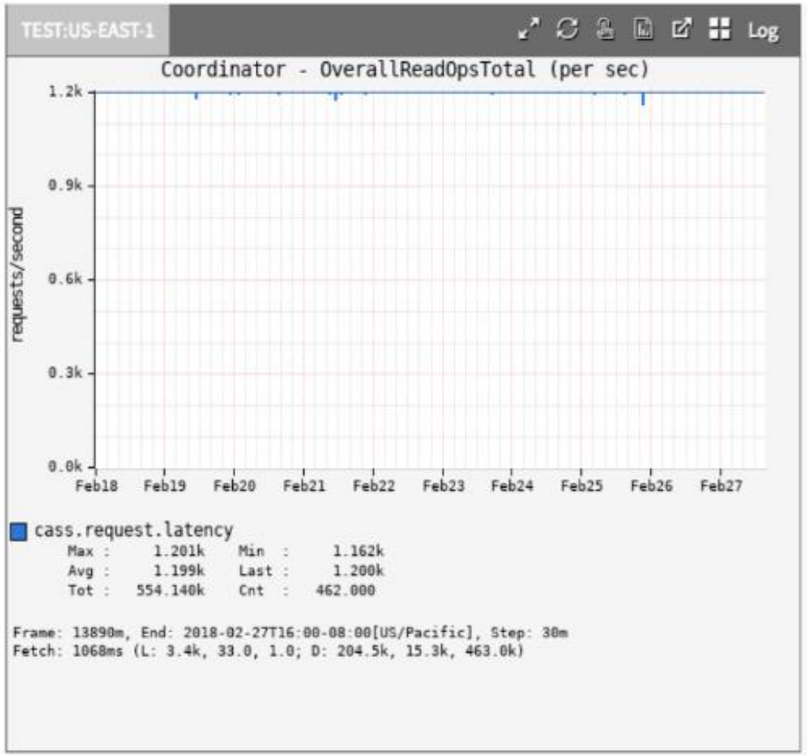
Module	API	Description	Query params
/driver	/initfromscript	Initializes dynamic plugin from script	
/driver	/startDataFill	Start the data fill process	
/driver	/startDataFillAsync	Start the data fill process asynchronously.	
/driver	/startConditionalDataFill	Starts the conditional data fill process.	
/driver	/startVerifyDataFill	Starts the verification for data fill process	
/driver	/stopDataFill	Stops data fill process	
/driver	/shutdownDataFill	stops data fill process	
/driver	/init/{client}	Initializing the client defined in {client}	client name
/driver	/start	Starts the NDBench test both read and write	
/driver	/startReads	Starts the NDBench read tests	
/driver	/stopReads	Stops the NDBench read tests	
/driver	/startWrites	Starts the NDBench write tests	
/driver	/stopWrites	Stops the NDBench read tests	
/driver	/stop	Stops the NDBench test both read and write	
/driver	/readSingle/{key}	Reads single key from initialized driver	key
/driver	/writeSingle/{key}	Writes single key to initizlied driver	key
/driver	/stats	Gets NDBench stats	
/driver	/getReadStatus	Gets NDBench read status	
/driver	/getWriteStatus	Gets NDBench write status	
/driver	/shutdownclient	disconnets and calls shudown process/ method on plugins. It also stop running reads and writes	
/driver	/getdrivers	Gets the list of drivers	
/driver	/getserverstatus	Returns a JSON for the status of the server	

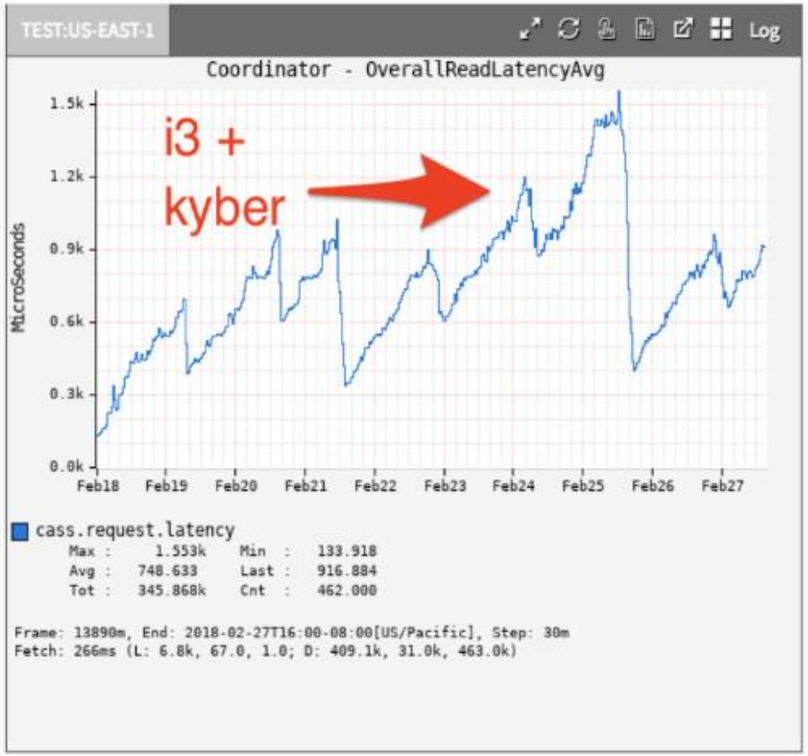
NDBench in action

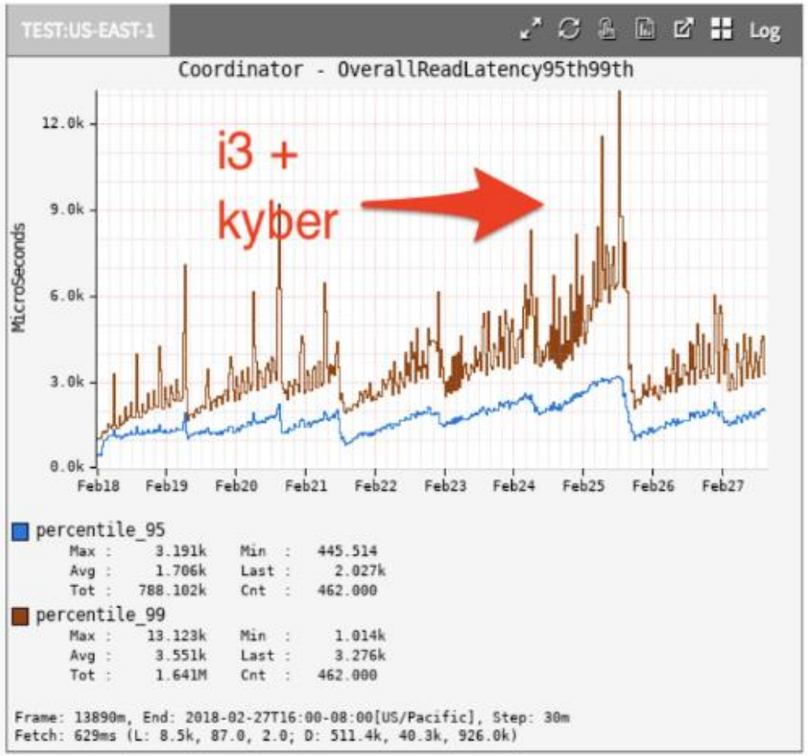
I2 vs I3 instance type



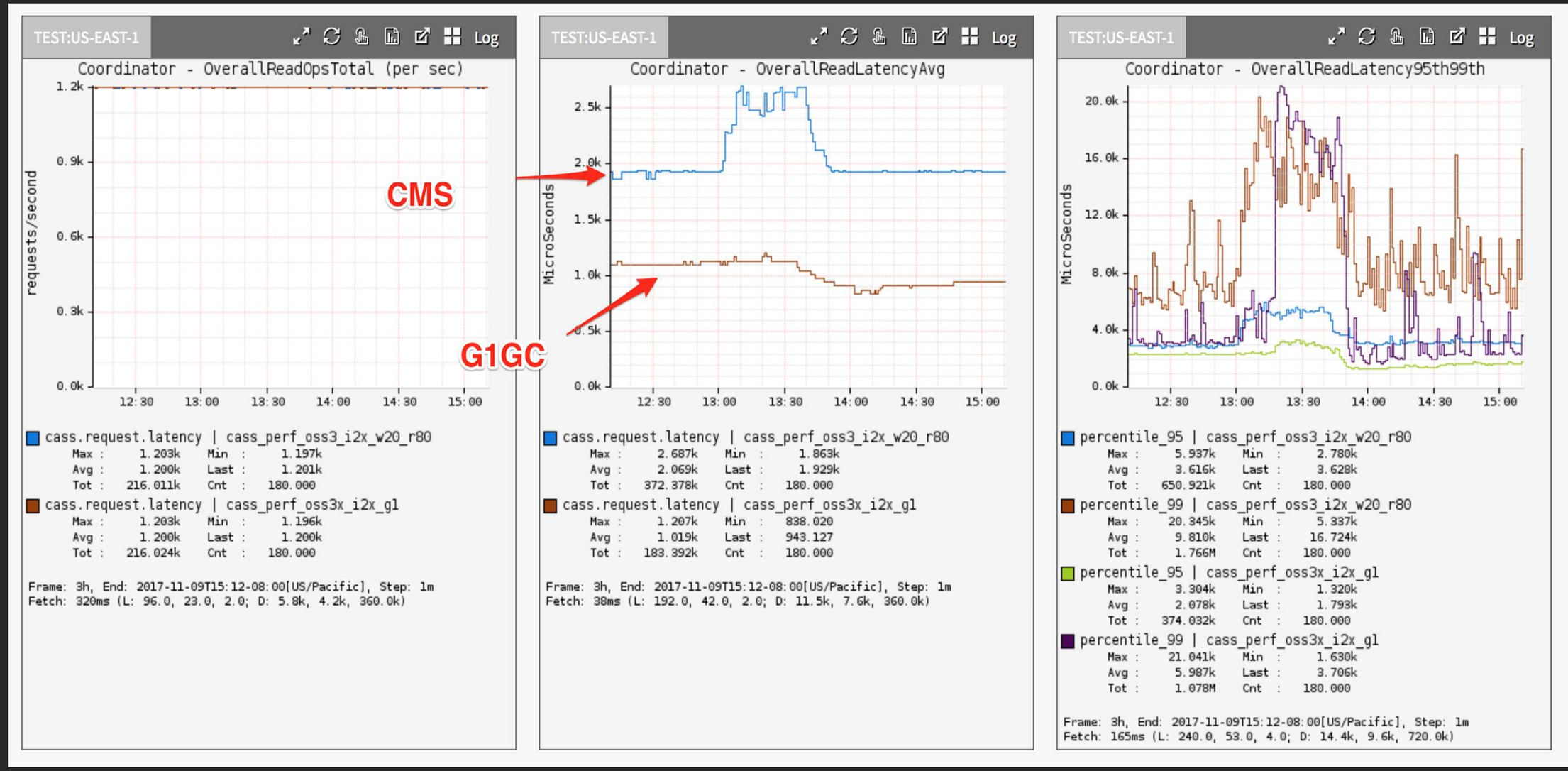
Cassandra performance on EC2 I3 instance with Kyber IO scheduler



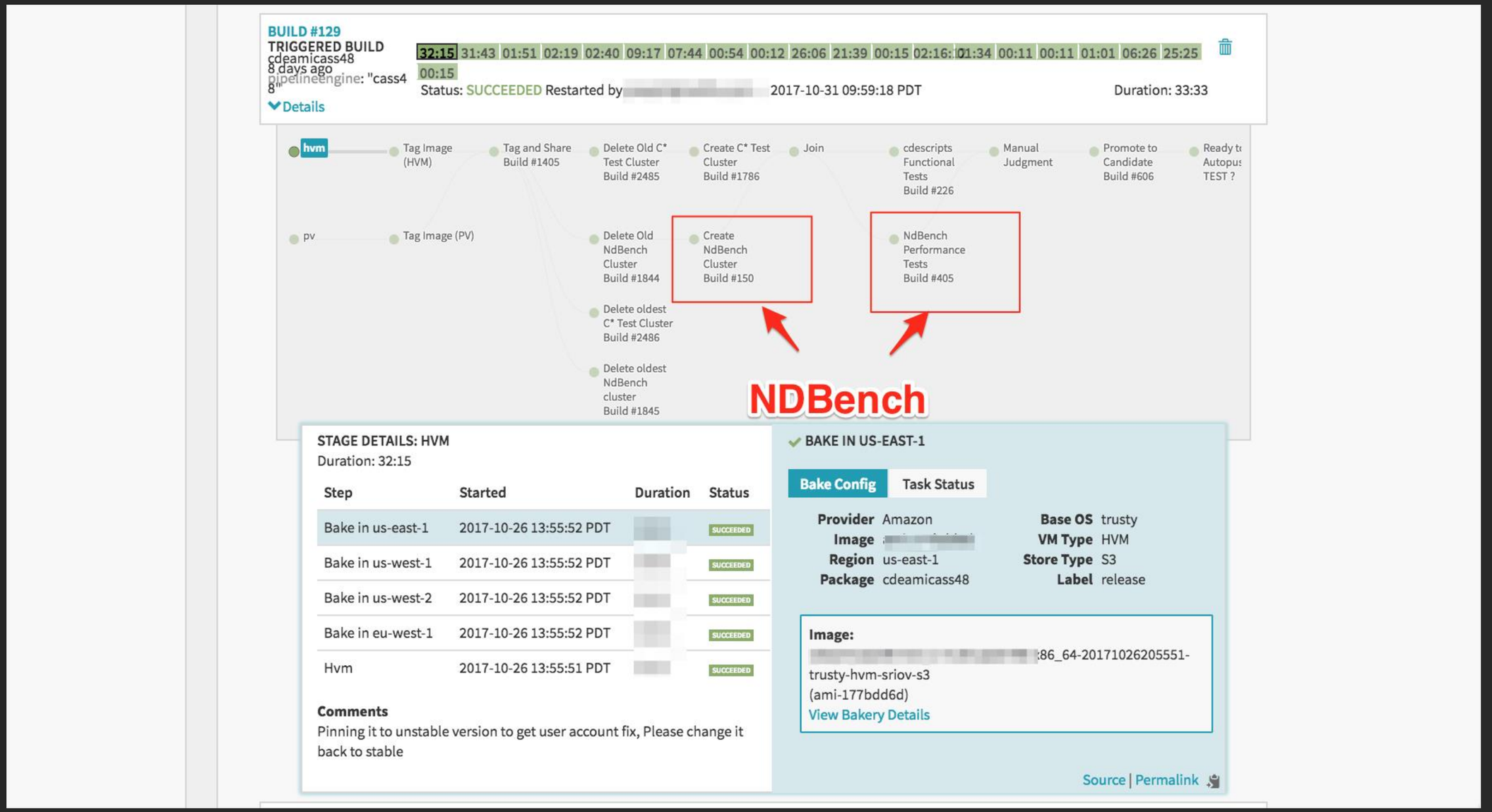




CMS vs G1GC for Cassandra



AMI certification process, integration and deployment validation



NDBench @ Netflix used for

- Active benchmarking system
- Building confidence on new services
- Continuous integration
- Deployment validation

Netflix benchmark philosophy – Tools

Active benchmarking

- NDBench
- **CI benchmarking**
- Preflight check

Passive benchmarking

- AWS bad IO detector
- jvmquake
- Declarative control planes

Active benchmarking

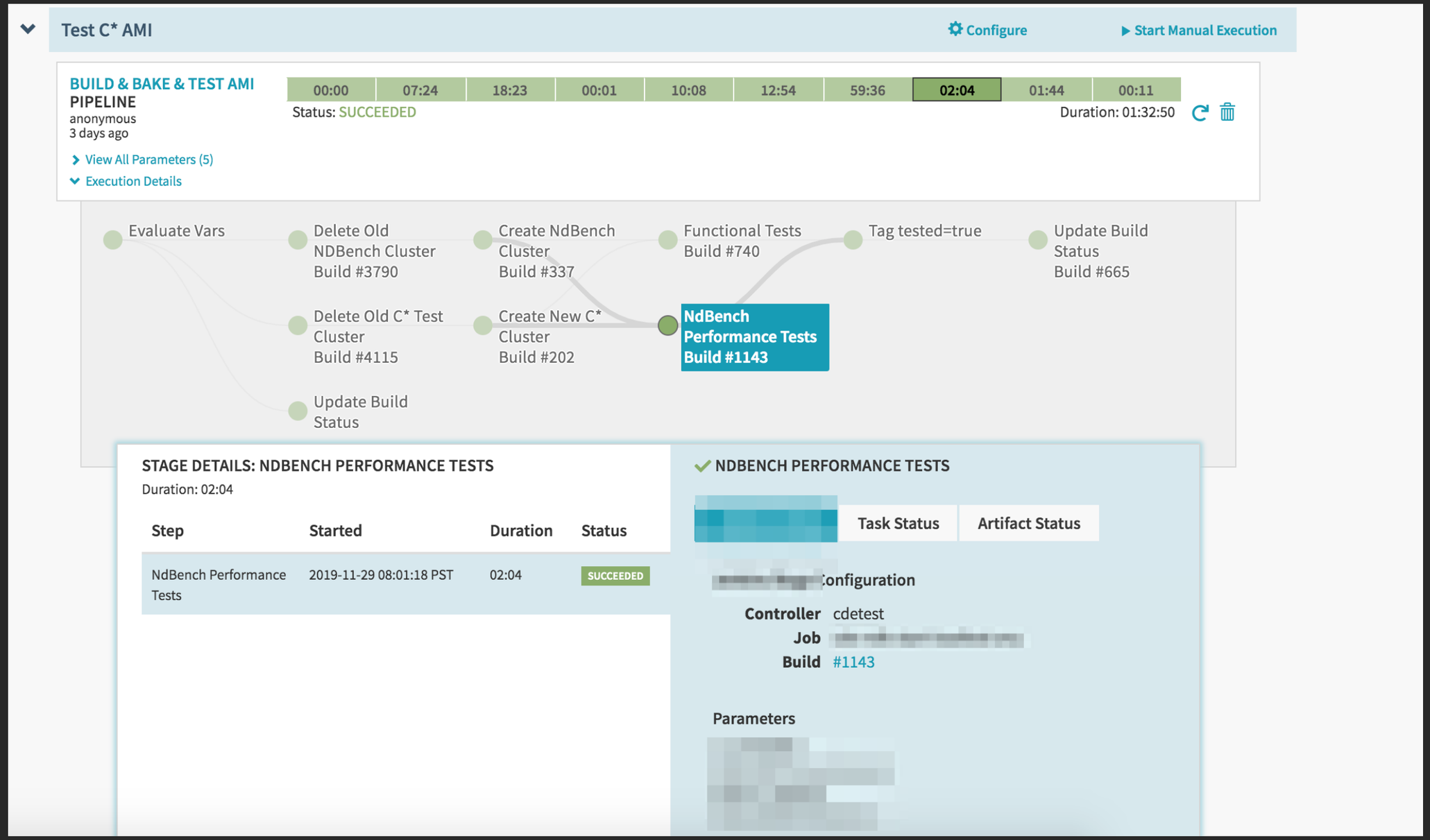
CI benchmarking

AWS re:Invent

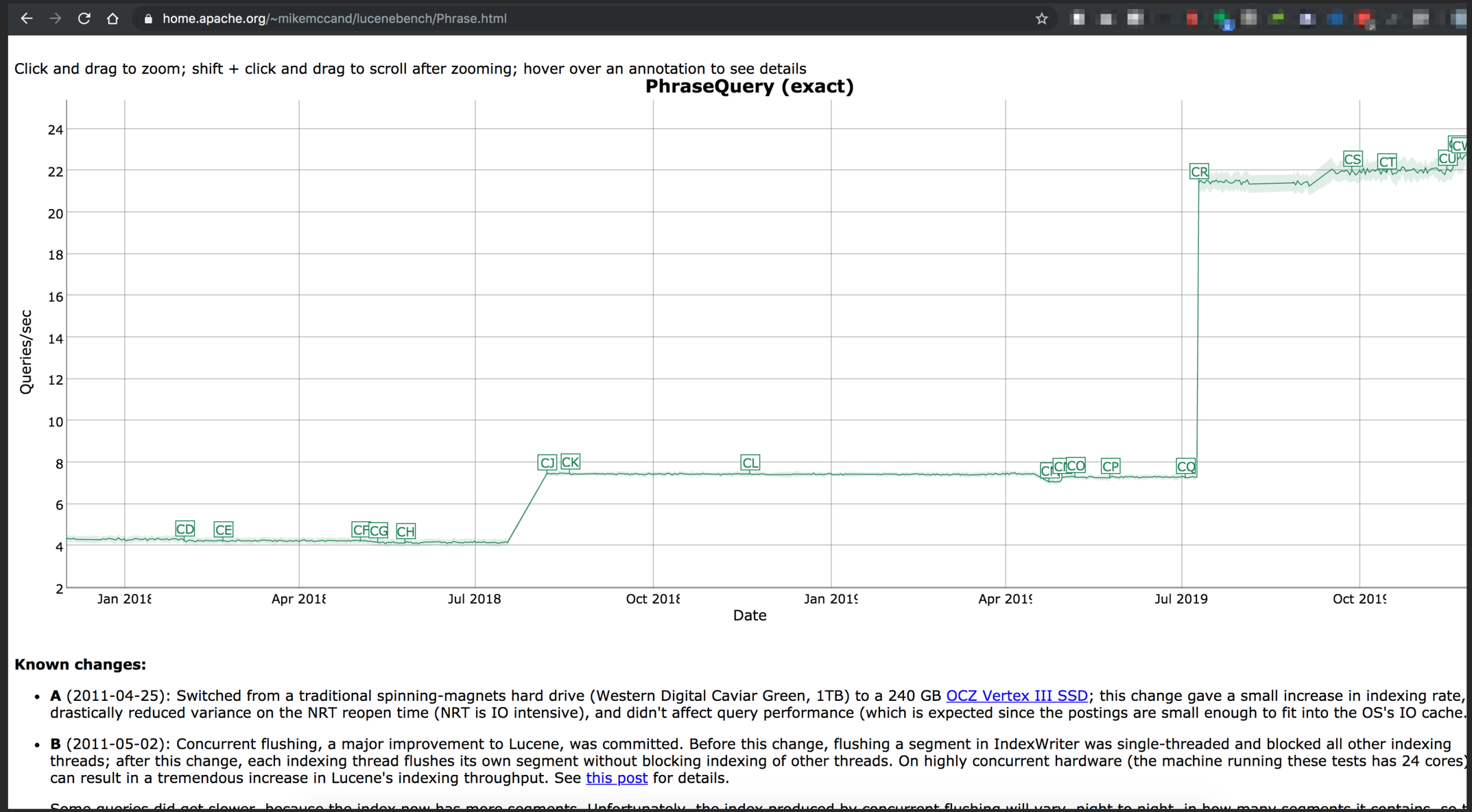
© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

NDBench in CI



Lucene benchmark – Open source way



Known changes:

- **A** (2011-04-25): Switched from a traditional spinning-magnets hard drive (Western Digital Caviar Green, 1TB) to a 240 GB OCZ Vertex III SSD; this change gave a small increase in indexing rate, drastically reduced variance on the NRT reopen time (NRT is IO intensive), and didn't affect query performance (which is expected since the postings are small enough to fit into the OS's IO cache.
- **B** (2011-05-02): Concurrent flushing, a major improvement to Lucene, was committed. Before this change, flushing a segment in IndexWriter was single-threaded and blocked all other indexing threads; after this change, each indexing thread flushes its own segment without blocking indexing of other threads. On highly concurrent hardware (the machine running these tests has 24 cores) can result in a tremendous increase in Lucene's indexing throughput. See this post for details.

Netflix benchmark philosophy - Tools

Active benchmarking

- NDBench
- CI benchmarking
- **Preflight check**

Passive benchmarking

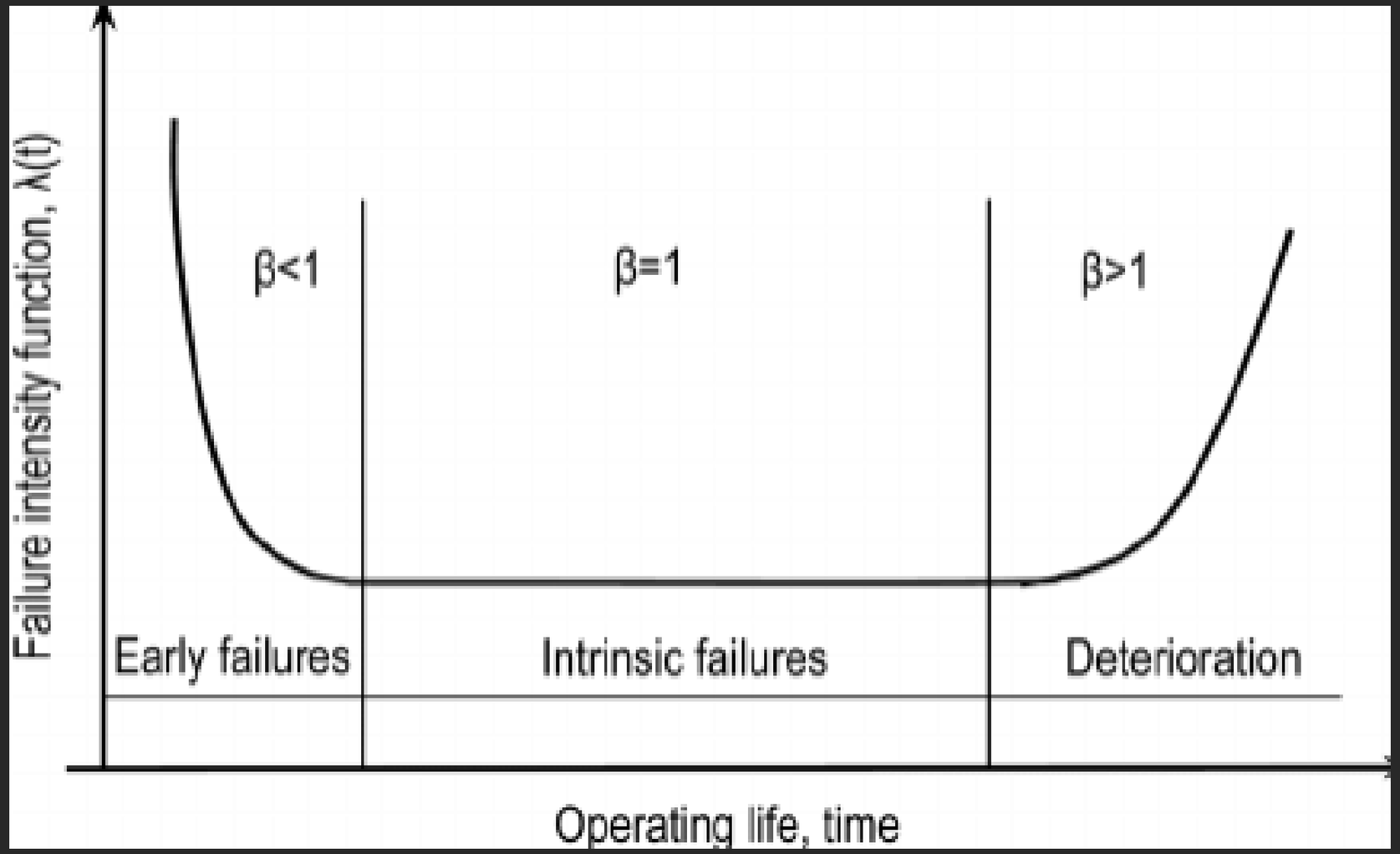
- AWS bad IO detector
- jvmquake
- Declarative control planes

Active benchmarking

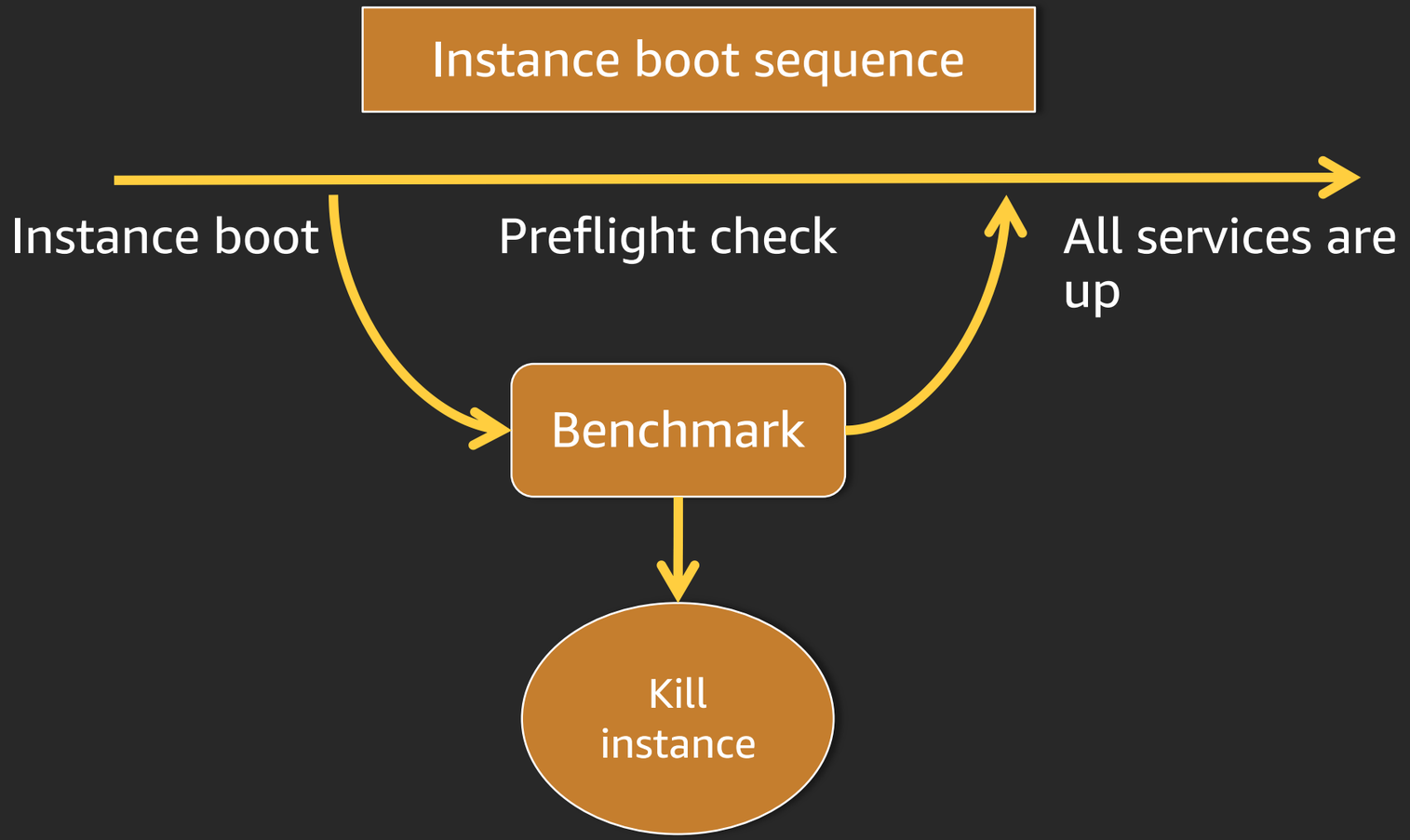
Preflight check

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Life of an instance



Preflight check



- Benchmark instances during launch/boot
- Do not promote instance to *InService* when benchmark fails
- It might delay your start time, but that is good

How about your disk during the flight?

Isn’t benchmarking code, config, data model, and instance during start enough?

Let’s take a look!

```
$ sudo dmesg | grep -i XFS
[   19.004581] SGI XFS with ACLs, security attributes, realtime,
no debug enabled
[   19.007941] XFS (md0): Mounting V5 Filesystem
[   19.146420] XFS (md0): Starting recovery (logdev: internal)
[   19.229059] XFS (md0): Ending recovery (logdev: internal)

$ cat /proc/mdstat
Personalities : [raid0] [linear] [multipath] [raid1] [raid6]
[raid5] [raid4] [raid10]
md0 : active raid0 nvme1n1[1] nvme0n1[0]
      3710675328 blocks super 1.1 64k chunks

unused devices: <none>
```

Healthy disk

```
$ sudo /usr/share/bcc/tools/biosnoop | grep nvme0n1 > ios
$ cat ios | tr -s ' ' | cut -f 8 -d ' ' | histogram.py
# NumSamples = 877; Min = 0.03; Max = 0.57
# Mean = 0.121209; Variance = 0.000891; SD = 0.029856; Median 0.120000
# each ■ represents a count of 10
    0.0300 -     0.0840 [     7]:
    0.0840 -     0.1380 [   764]: ■■■■■■■■■■■■■■■■■■■■■■■■■■
    0.1380 -     0.1920 [   102]: ■■■■■■■■■■
    0.1920 -     0.2460 [     1]:
    0.2460 -     0.3000 [     0]:
    0.3000 -     0.3540 [     0]:
    0.3540 -     0.4080 [     0]:
    0.4080 -     0.4620 [     0]:
    0.4620 -     0.5160 [     1]:
    0.5160 -     0.5700 [     2]:
```

This is what I3s should look like

```
$ sudo /usr/share/bcc/tools/biosnoop | grep nvme0n1 > ios
$ cat ios | tr -s ' ' | cut -f 8 -d ' ' | histogram.py
# NumSamples = 10000; Min = 0.08; Max = 695.35
# Mean = 8.908690; Variance = 3680.9398; SD = 60.6707; Median 0.120000
# each ■ represents a count of 96
    0.0800 -     0.7596 [  7260]: ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    0.7596 -     2.1189 [   101]: ■
    2.1189 -     4.8375 [   120]: ■
    4.8375 -    10.2746 [   454]: ■■■■
   10.2746 -    21.1488 [  1922]: ■■■■■■■■■■■■■■■■■■■■
   21.1488 -    42.8972 [    58]:
   42.8972 -    86.3941 [     0]:
   86.3941 -   173.3878 [     0]:
  173.3878 -   347.3752 [     0]:
  347.3752 -   695.3500 [    85]:
```

This does *not* look fine

Let’s transfer the data
and get a new node!

If only it were easy!

Passive benchmarking to the rescue!

Netflix benchmark philosophy – Tools

Active benchmarking

- NDBench
- CI benchmarking
- Preflight check

Passive benchmarking

- **AWS bad IO detector**
- jvmquake
- Declarative control planes

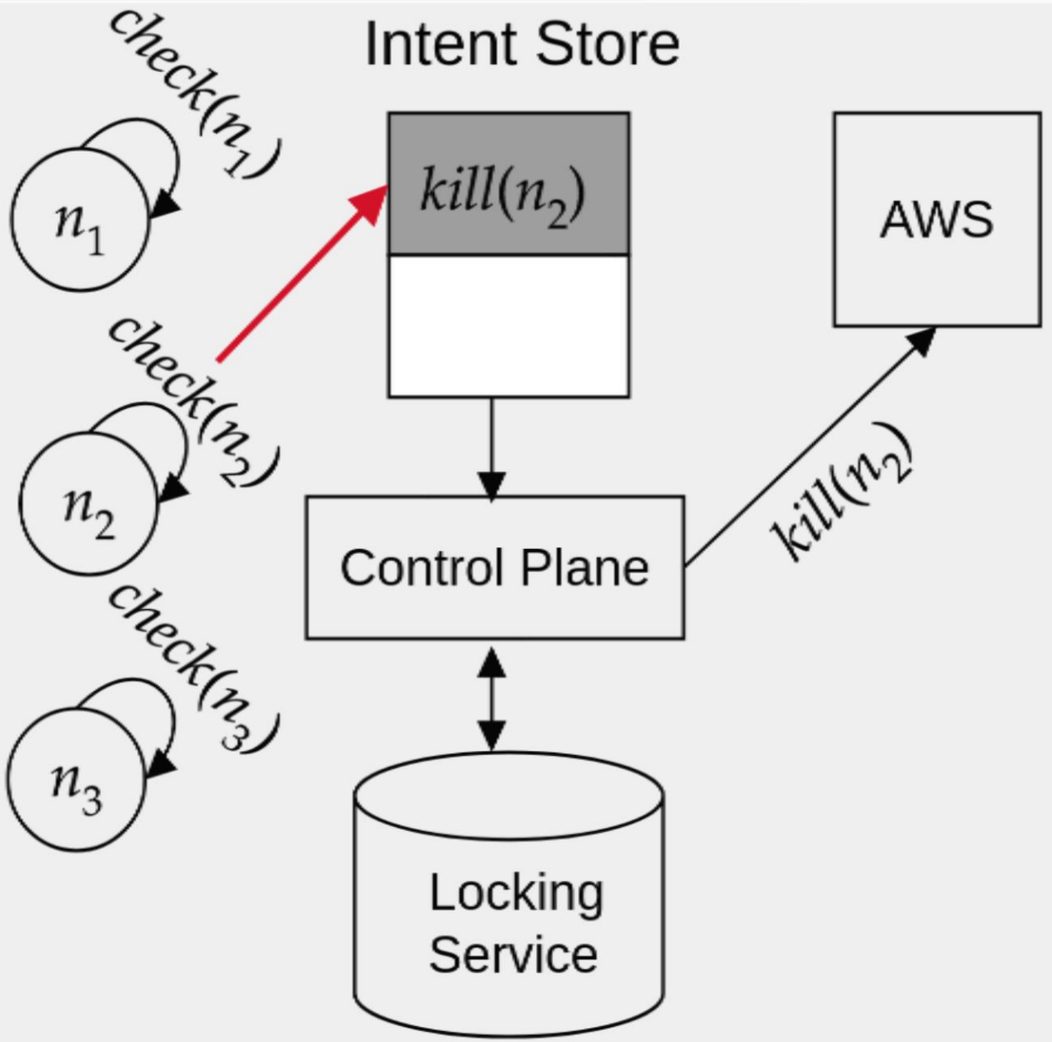
Passive benchmarking

AWS bad IO detector

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

AWS bad IO detector

- Desire: Degraded hardware should die <10m
- Observe: Local disks, creds, filesystems
- Action: Request termination



$$
\begin{aligned}
check(n_i) \;=\; (& \\
& fs_failed(n_i) && \vee \\
& |ioerror(n_i)| \;>\; 5 && \vee \\
& \neg write(/\,\mathsf{mnt},\; n_i)) && \vee \\
& latency_{disk}(n_i) \;>\; 100ms & \\
) &
\end{aligned}
$$

$$
check(n_i) \;\rightarrow\; kill(n_i)
$$

Netflix benchmark philosophy – Tools

Active benchmarking

- NDBench
- CI benchmarking
- Preflight check

Passive benchmarking

- AWS bad IO detector
- **jvmquake**
- Declarative control planes

Passive benchmarking

jvmquake

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

GC spiral of death

Operating JVM-based stateful services is fun and challenging!

JVM introduces new variables in your system

- How to predict and benchmark JVM with production load?



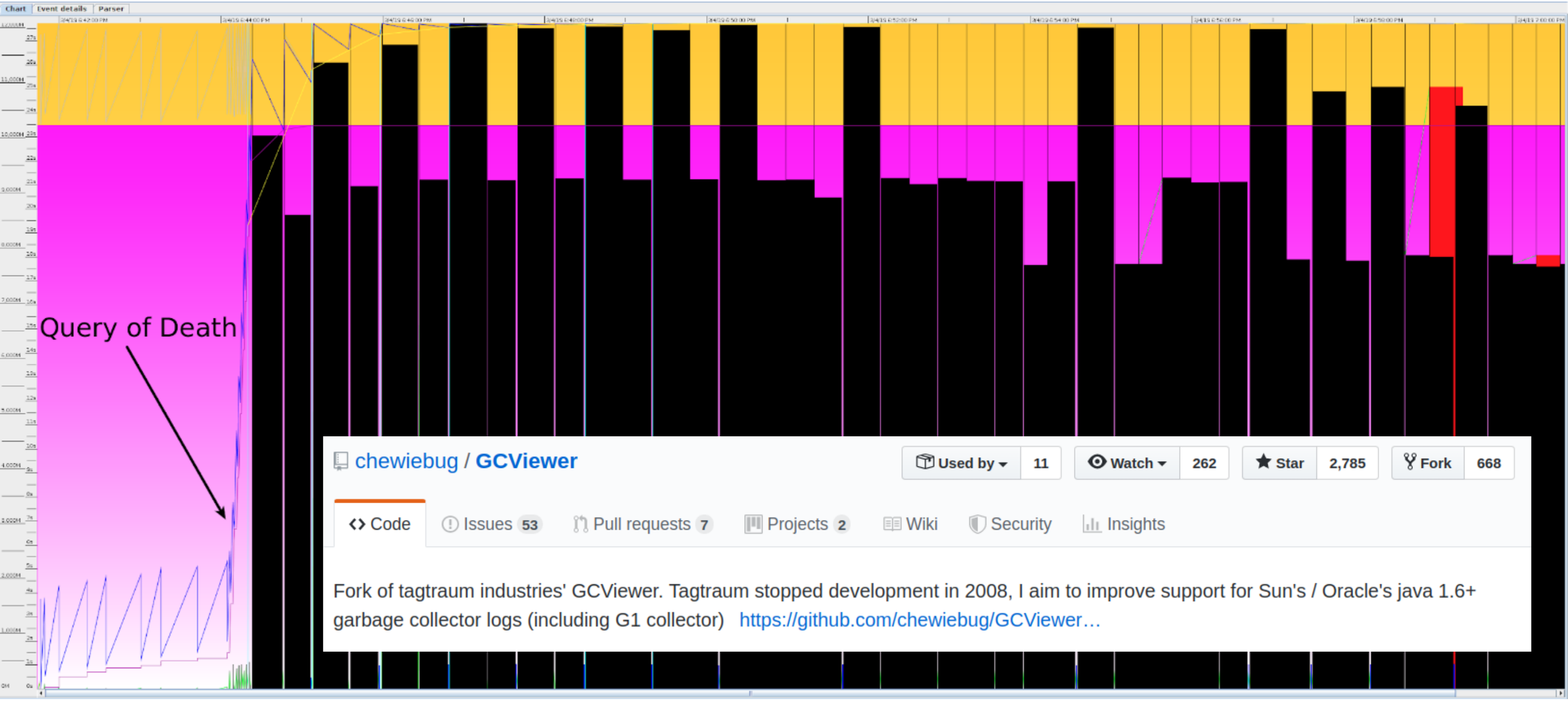


“Safepoint” Pauses -> Disks

```
$ grep "stopped" gclog | tail
2019-03-05T02:59:15.111+0000: 1057.680: Total time for which application threads were
stopped: 18.0113457 seconds, Stopping threads took: 0.0001798 seconds
2019-03-05T02:59:16.159+0000: 1058.728: Total time for which application threads were
stopped: 1.0472826 seconds, Stopping threads took: 0.0001542 seconds
2019-03-05T02:59:40.461+0000: 1083.030: Total time for which application threads were
stopped: 24.3016592 seconds, Stopping threads took: 0.0001809 seconds
2019-03-05T03:00:16.263+0000: 1118.832: Total time for which application threads were
stopped: 35.8020625 seconds, Stopping threads took: 0.0001307 seconds
2019-03-05T03:00:51.599+0000: 1154.168: Total time for which application threads were
stopped: 35.3361231 seconds, Stopping threads took: 0.0001517 seconds
2019-03-05T03:01:09.470+0000: 1172.039: Total time for which application threads were
stopped: 17.8703301 seconds, Stopping threads took: 0.0002151 seconds
```

GC Pauses -> JVM

20-second GC pauses … not good


Chart
Event details
Parser
Query of Death
chewiebug / GCViewer
Used by 11
Watch 262
Star 2,785
Fork 668
Code
Issues 53
Pull requests 7
Projects 2
Wiki
Security
Insights
Fork of tagtraum industries' GCViewer. Tagtraum stopped development in 2008, I aim to improve support for Sun's / Oracle's java 1.6+ garbage collector logs (including G1 collector) https://github.com/chewiebug/GCViewer…

This is bad

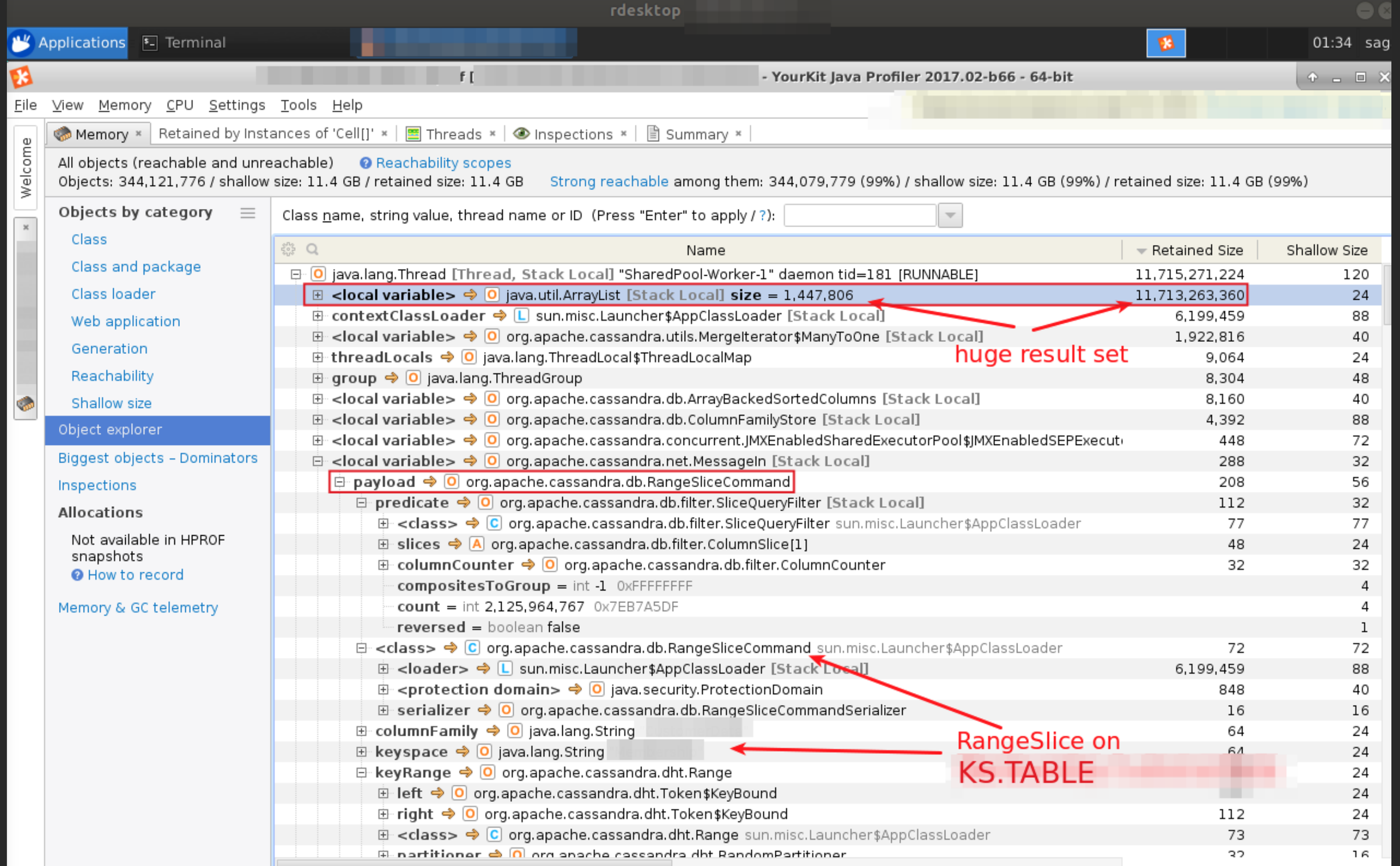
```
$ jstat -gcutil $(pgrep -f Cassandra) 1s
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
100.00   0.00 100.00 100.00  97.79  94.95  14042  624.634    10   88.798  713.432
100.00   0.00 100.00 100.00  97.79  94.95  14042  624.634    10   88.798  713.432
100.00   0.00 100.00 100.00  97.79  94.95  14042  624.634    10   88.798  713.432

# Might have to run this in a while loop if you're really stuck ...
$ sudo -u cassandra jmap -dump:live,format=b,file=core.hprof $(pgrep -f Cassandra)
Dumping heap to /mnt/data/cassandra/dumps/core.hprof …

# Since it will be O(heap size) large, you probably need to take it off box
$ aws s3 cp core.hprof s3://bucket/to/stash/coredumps

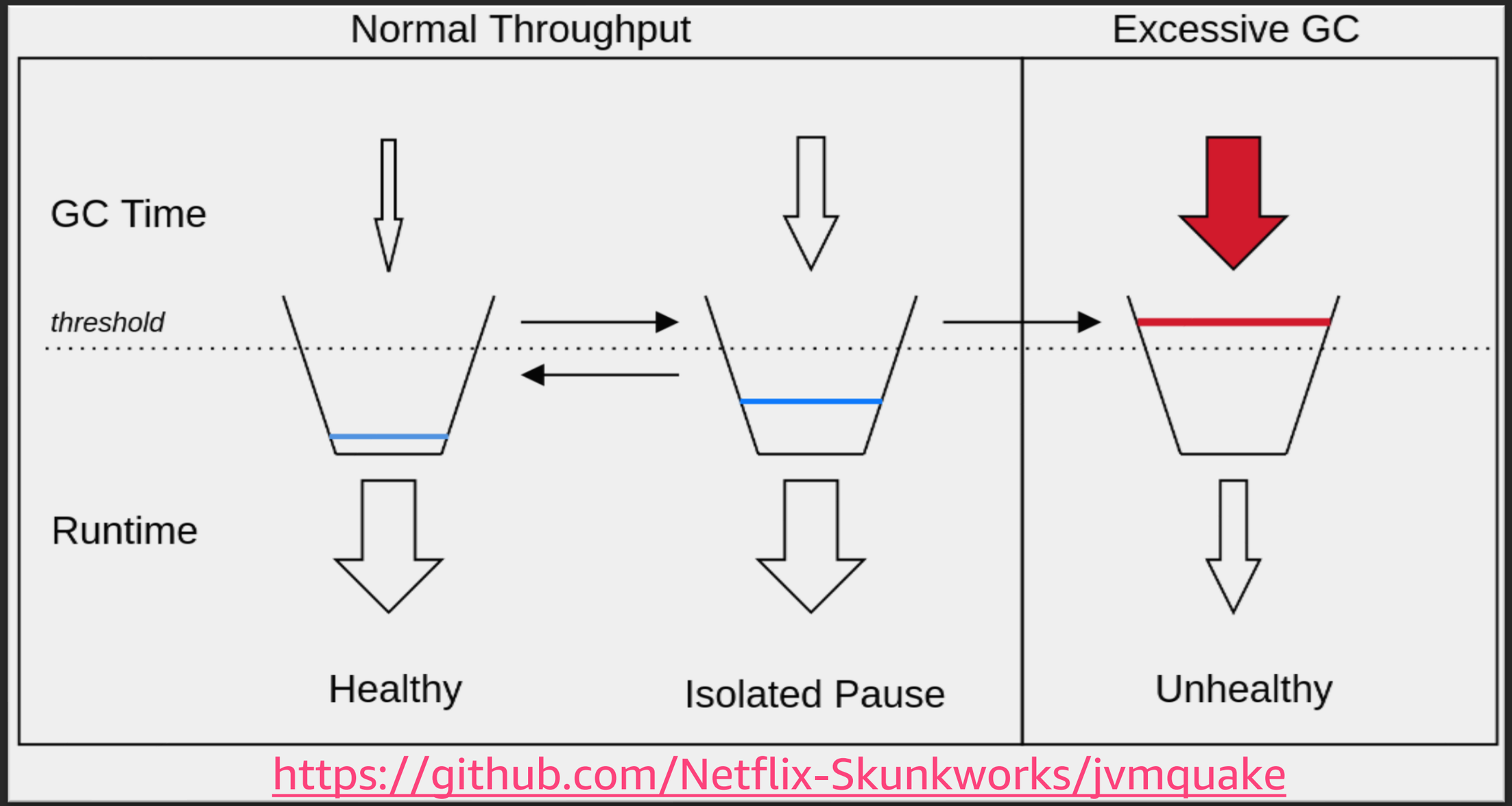
# Mitigate, probably need to forcibly kill since C* has a sigterm handler
$ sudo kill -9 $(pgrep -f Cassandra)
```

100% heap utilization is bad


rdesktop
Applications
Terminal
01:34 sag
- YourKit Java Profiler 2017.02-b66 - 64-bit
File View Memory CPU Settings Tools Help
Memory
Retained by Instances of 'Cell[]'
Threads
Inspections
Summary
Welcome
All objects (reachable and unreachable)
Reachability scopes
Objects: 344,121,776 / shallow size: 11.4 GB / retained size: 11.4 GB
Strong reachable among them: 344,079,779 (99%) / shallow size: 11.4 GB (99%) / retained size: 11.4 GB (99%)
Objects by category
Class
Class and package
Class loader
Web application
Generation
Reachability
Shallow size
Object explorer
Biggest objects – Dominators
Inspections
Allocations
Not available in HPROF snapshots
How to record
Memory & GC telemetry
Class name, string value, thread name or ID (Press "Enter" to apply / ?):
Name
Retained Size
Shallow Size
java.lang.Thread [Thread, Stack Local] "SharedPool-Worker-1" daemon tid=181 [RUNNABLE] 11,715,271,224 120
<local variable> java.util.ArrayList [Stack Local] size = 1,447,806 11,713,263,360 24
contextClassLoader sun.misc.Launcher$AppClassLoader [Stack Local] 6,199,459 88
<local variable> org.apache.cassandra.utils.MergeIterator$ManyToOne [Stack Local] 1,922,816 40
threadLocals java.lang.ThreadLocal$ThreadLocalMap 9,064 24
group java.lang.ThreadGroup 8,304 48
<local variable> org.apache.cassandra.db.ArrayBackedSortedColumns [Stack Local] 8,160 40
<local variable> org.apache.cassandra.db.ColumnFamilyStore [Stack Local] 4,392 88
<local variable> org.apache.cassandra.concurrent.JMXEnabledSharedExecutorPool$JMXEnabledSEPExecut 448 72
<local variable> org.apache.cassandra.net.MessageIn [Stack Local] 288 32
payload org.apache.cassandra.db.RangeSliceCommand 208 56
predicate org.apache.cassandra.db.filter.SliceQueryFilter [Stack Local] 112 32
<class> org.apache.cassandra.db.filter.SliceQueryFilter sun.misc.Launcher$AppClassLoader 77 77
slices org.apache.cassandra.db.filter.ColumnSlice[1] 48 24
columnCounter org.apache.cassandra.db.filter.ColumnCounter 32 32
compositesToGroup = int -1 0xFFFFFFFF 4
count = int 2,125,964,767 0x7EB7A5DF 4
reversed = boolean false 1
<class> org.apache.cassandra.db.RangeSliceCommand sun.misc.Launcher$AppClassLoader 72 72
<loader> sun.misc.Launcher$AppClassLoader [Stack Local] 6,199,459 88
<protection domain> java.security.ProtectionDomain 848 40
serializer org.apache.cassandra.db.RangeSliceCommandSerializer 16 16
columnFamily java.lang.String 64 24
keyspace java.lang.String 64 24
keyRange org.apache.cassandra.dht.Range 24
left org.apache.cassandra.dht.Token$KeyBound 24
right org.apache.cassandra.dht.Token$KeyBound 112 24
<class> org.apache.cassandra.dht.Range sun.misc.Launcher$AppClassLoader 73 73
huge result set
RangeSlice on
KS.TABLE

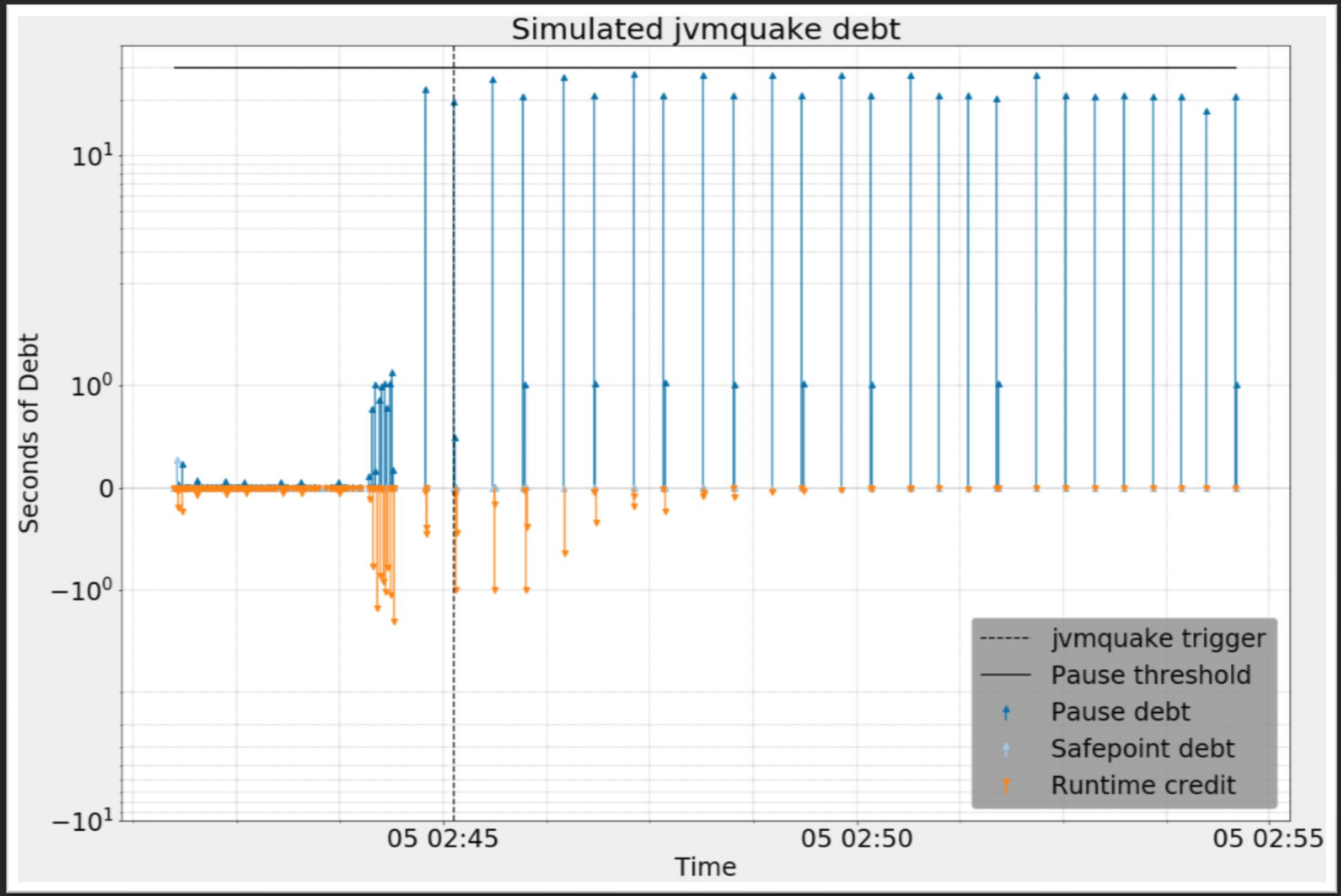
But . . . it is useful only when it is automated!

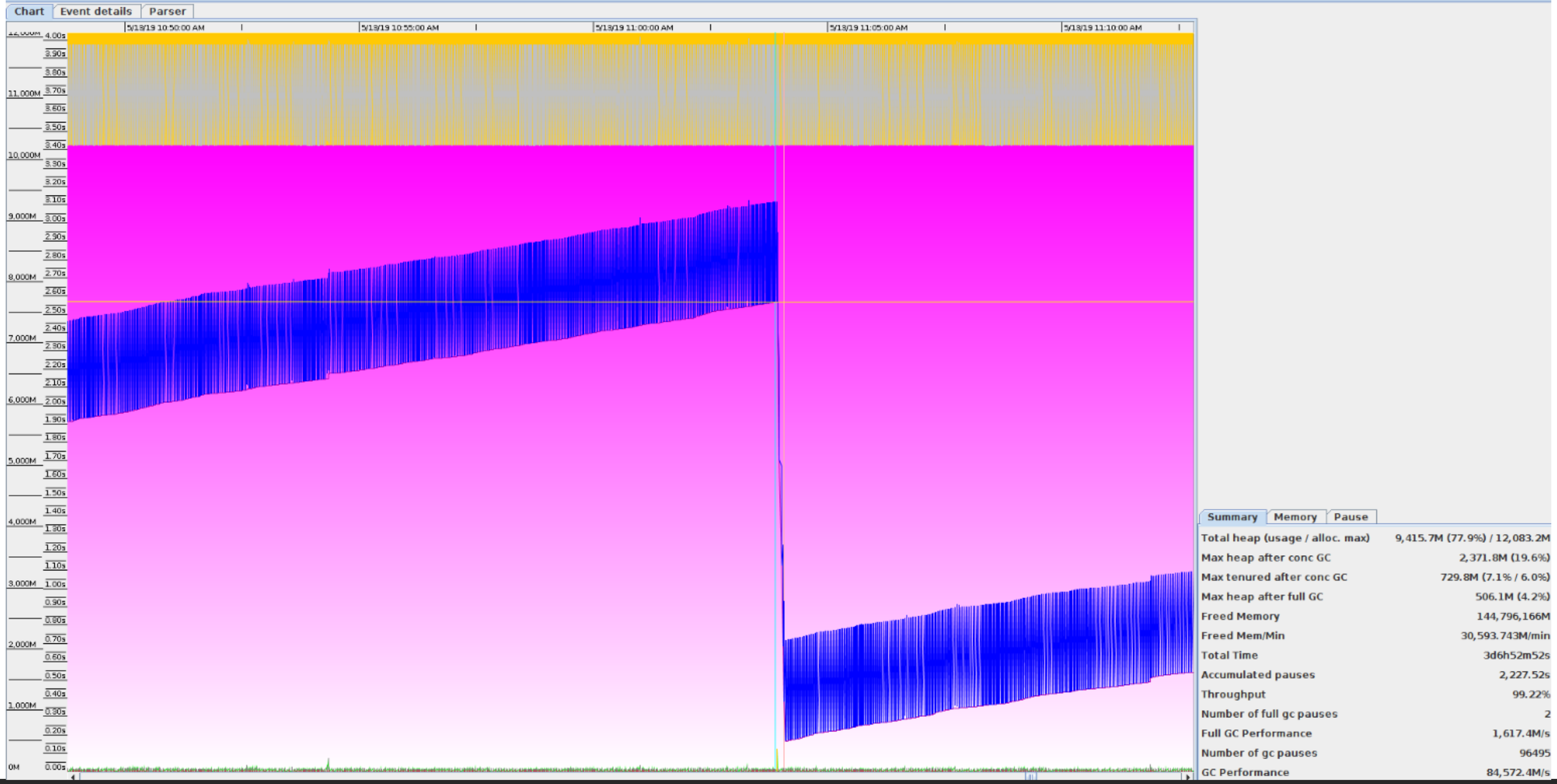
Automate – GC spiral of death



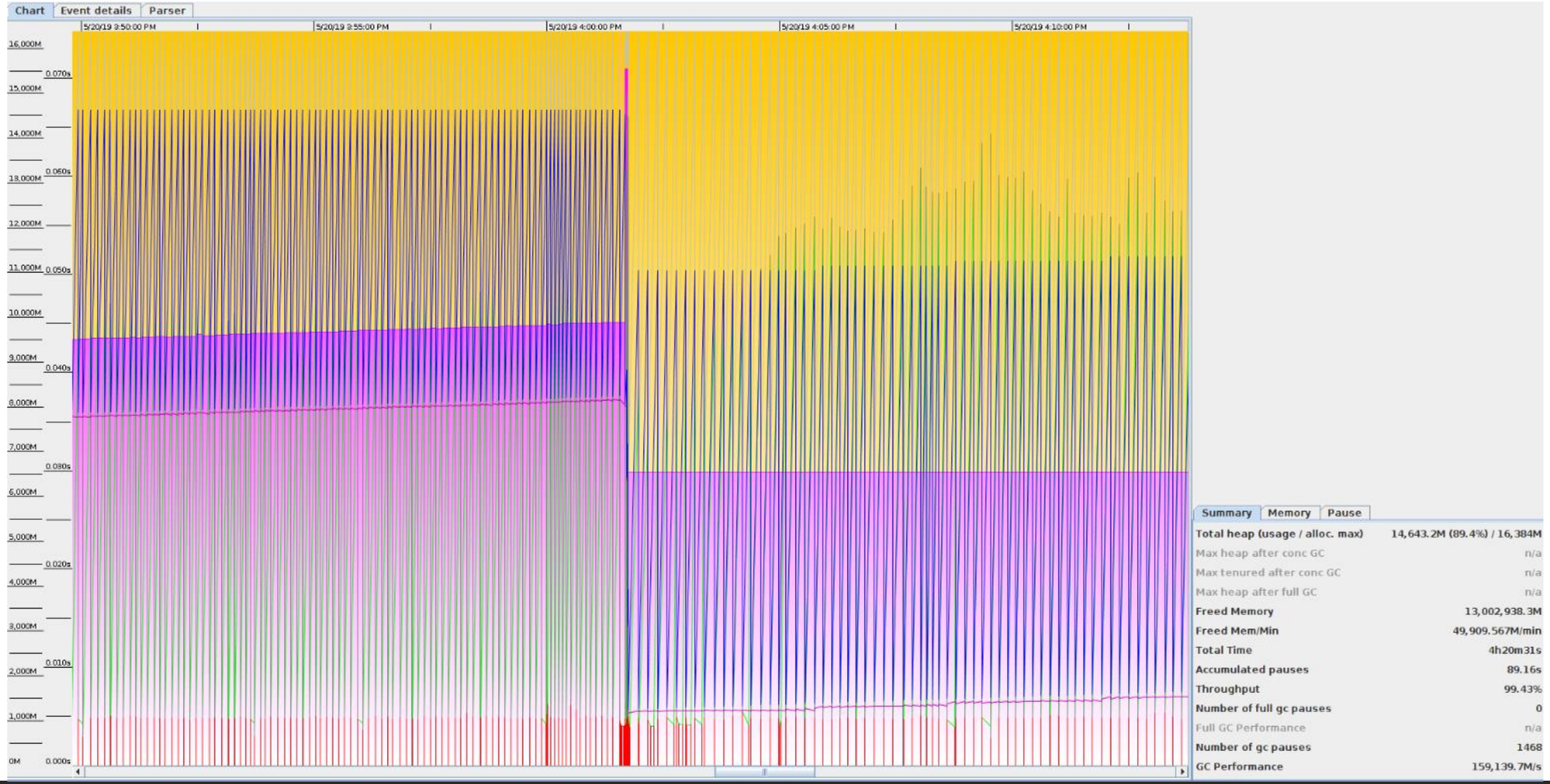
https://github.com/Netflix-Skunkworks/jvmquake

Automate - GC spiral of death





This is normal CMS



This is normal G1GC

Summary: Netflix benchmark philosophy – Tools

Active benchmarking

- NDBench
- CI benchmarking
- Preflight check

Passive benchmarking

- AWS bad IO detector
- jvmquake

A few case studies

- Amazon Elastic Block Store (Amazon EBS)
- Amazon DynamoDB
- Amazon S3
- Distributed caches

Benchmarking Amazon EBS for mission critical databases

- Majority of stateful services run on I3 instance family
- But ephemeral storage does not scale!
- How about Amazon EBS?
  - But which one though?
  - Each Amazon EBS product has different tradeoffs
- EBS-optimized instance (i3.2xl)
  - 212.5MB/s of aggregate bandwidth to EBS
  - 250MB/s per volume (gp2)
  - 3 IOPS per GB (gp2)

Getting Amazon EBS to do the job

- Latency
  - Instance store (I3) - RTT of 20µs
  - Amazon EBS – 130µs for reads/370µs for writes
- Workload generation is a key challenge
  - Sizing the workload
  - Unexpected clustering of reads and writes

Read-only Cassandra workload on Amazon EBS

```
# NumSamples = 36328; Min = 0.20; Max = 11.56
# Mean = 1.014655; Variance = 0.394151; SD = 0.627815; Median 0.862000
# each ■ represents a count of 350
    0.2050 -     0.2050 [     1]:
    0.2050 -     0.2050 [     0]:
    0.2050 -     0.2051 [     0]:
    0.2051 -     0.2052 [     0]:
    0.2052 -     0.2053 [     0]:
    0.2053 -     0.2057 [     0]:
    0.2057 -     0.2064 [     0]:
    0.2064 -     0.2078 [     0]:
    0.2078 -     0.2105 [     0]:
    0.2105 -     0.2161 [     1]:
    0.2161 -     0.2272 [     0]:
    0.2272 -     0.2494 [     1]:
    0.2494 -     0.2937 [     0]:
    0.2937 -     0.3825 [   197]:
    0.3825 -     0.5600 [   912]: ■■
    0.5600 -     0.9150 [ 26303]:
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    0.9150 -     1.6250 [  5809]: ■■■■■■■■■■■■■■■■
    1.6250 -     3.0450 [  2138]: ■■■■■■
    3.0450 -     5.8850 [   937]: ■■
    5.8850 -    11.5650 [    29]:
```

Data was collected with iosnoop | grep ' R ‘. Units – Milliseconds

Read-only Cassandra workload on instance store

```
# NumSamples = 186869; Min = 0.02; Max = 9.34
# Mean = 0.083496; Variance = 0.005143; SD = 0.071713; Median 0.064000
# each ■ represents a count of 1273
    0.0190 -     0.0190 [     2]:
    0.0190 -     0.0190 [     0]:
    0.0190 -     0.0191 [     0]:
    0.0191 -     0.0191 [     0]:
    0.0191 -     0.0193 [     0]:
    0.0193 -     0.0196 [     0]:
    0.0196 -     0.0201 [     2]:
    0.0201 -     0.0213 [     7]:
    0.0213 -     0.0235 [   131]:
    0.0235 -     0.0281 [  3184]: ■■
    0.0281 -     0.0372 [  5994]: ■■■■
    0.0372 -     0.0554 [ 22965]: ■■■■■■■■■■■■■■■■■■
    0.0554 -     0.0918 [ 95519]:
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    0.0918 -     0.1646 [ 54687]: ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    0.1646 -     0.3101 [  3717]: ■■
    0.3101 -     0.6012 [   471]:
    0.6012 -     1.1835 [   124]:
    1.1835 -     2.3480 [    31]:
    2.3480 -     4.6770 [    29]:
    4.6770 -     9.3350 [     6]:
```

Data was collected with iosnoop | grep ' R ‘. Units – Milliseconds

Well, let’s understand Amazon EBS and tune our workload

readahead

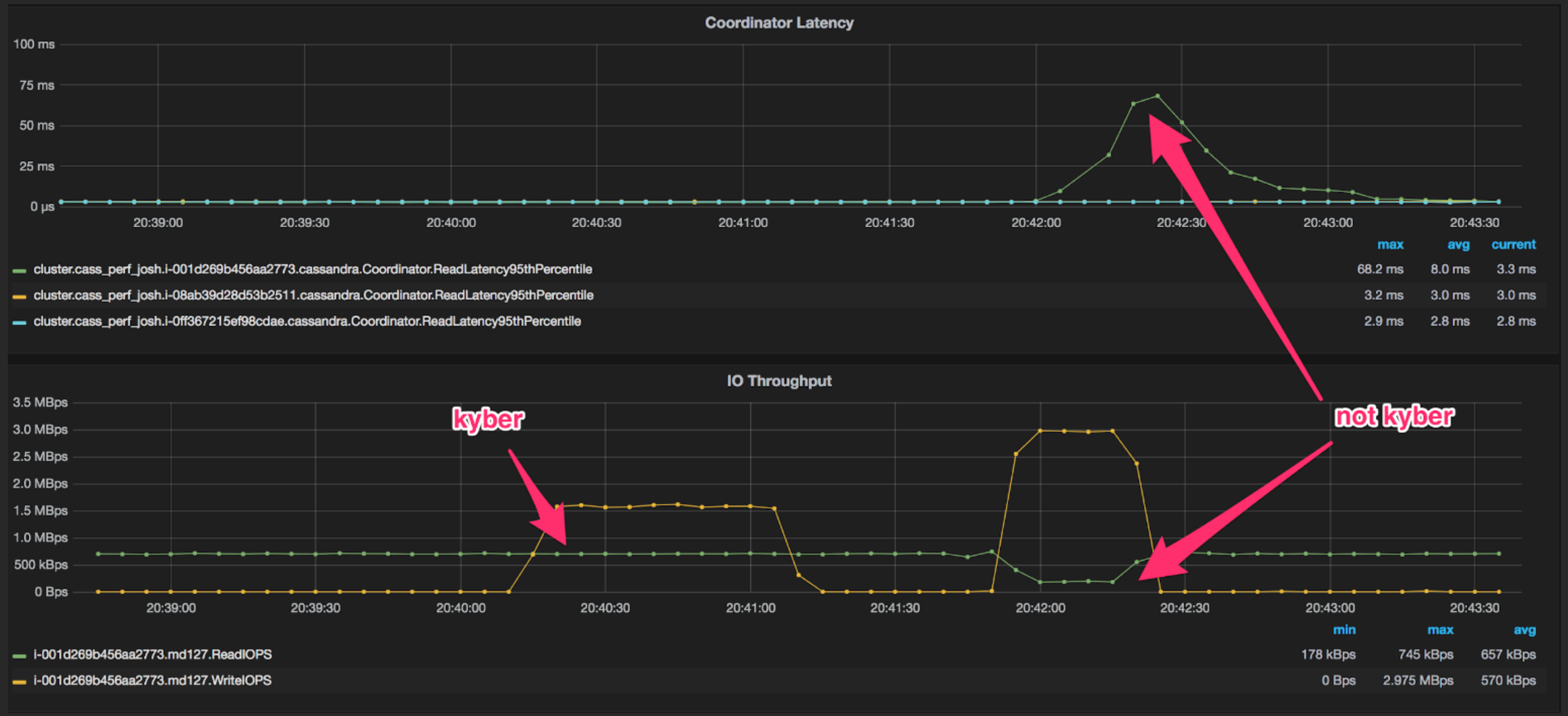
disk scheduling

working set size

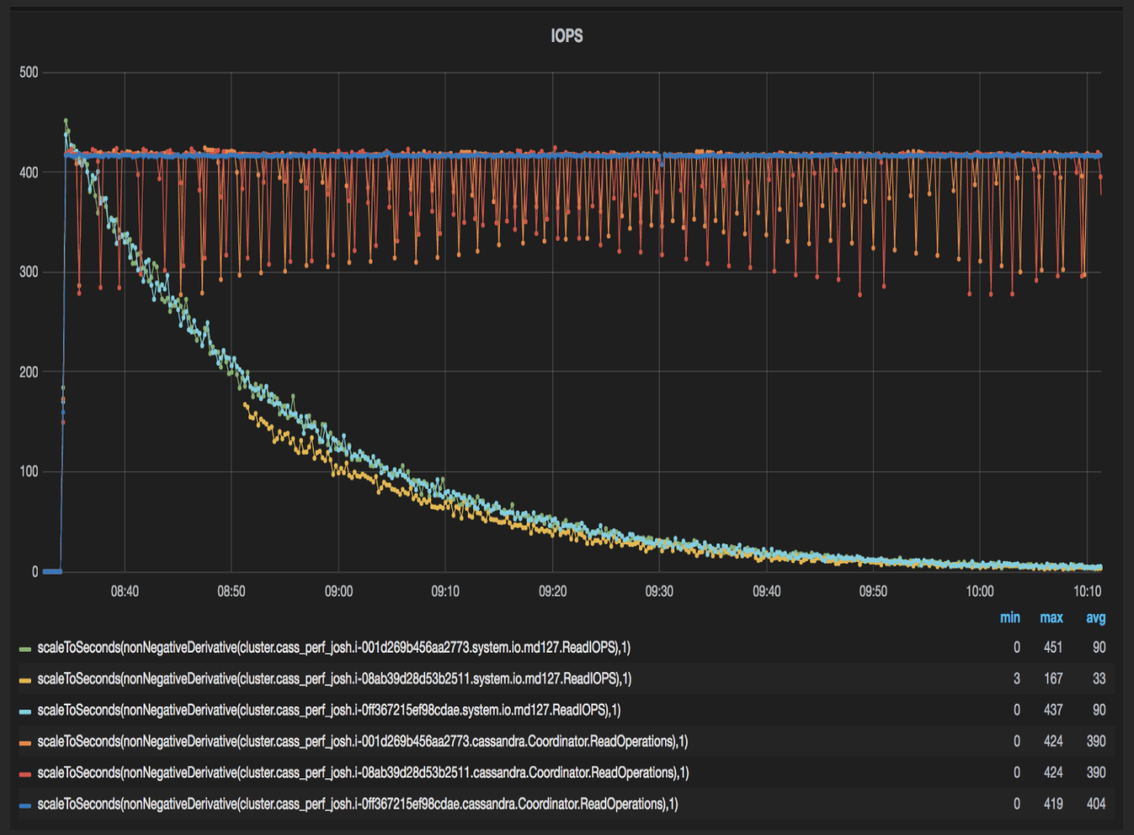
readahead

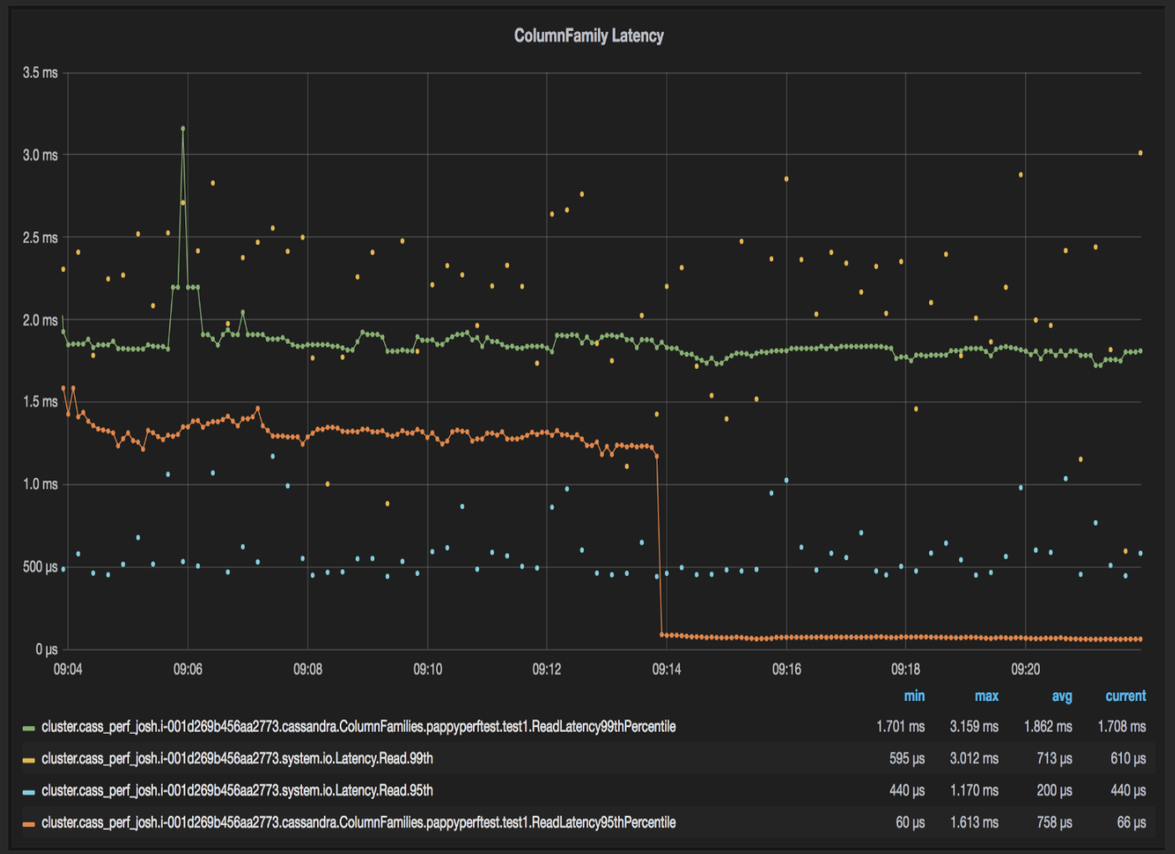
With readahead of 32kB, Cassandra compaction threads on GP2 can reach `compaction_throughput_mb_per_sec` of 128MB/s

disk scheduling



working set size





Lessons learned

Latency

- Average response times on Amazon EBS vs. I3 are not comparable
- 95th latencies are better
- 99th percentiles are very well in alignment

Lots of system CPU usage

Compaction strategy plays a major role

Conclusions

Size your compute and storage independently

Reduce management overhead

Use it for

- Storage dense
- Heavily cached

Identify the bottlenecks; something being faster or slower doesn’t really matter

Mixture of active and passive benchmarking

Benchmarking Amazon DynamoDB

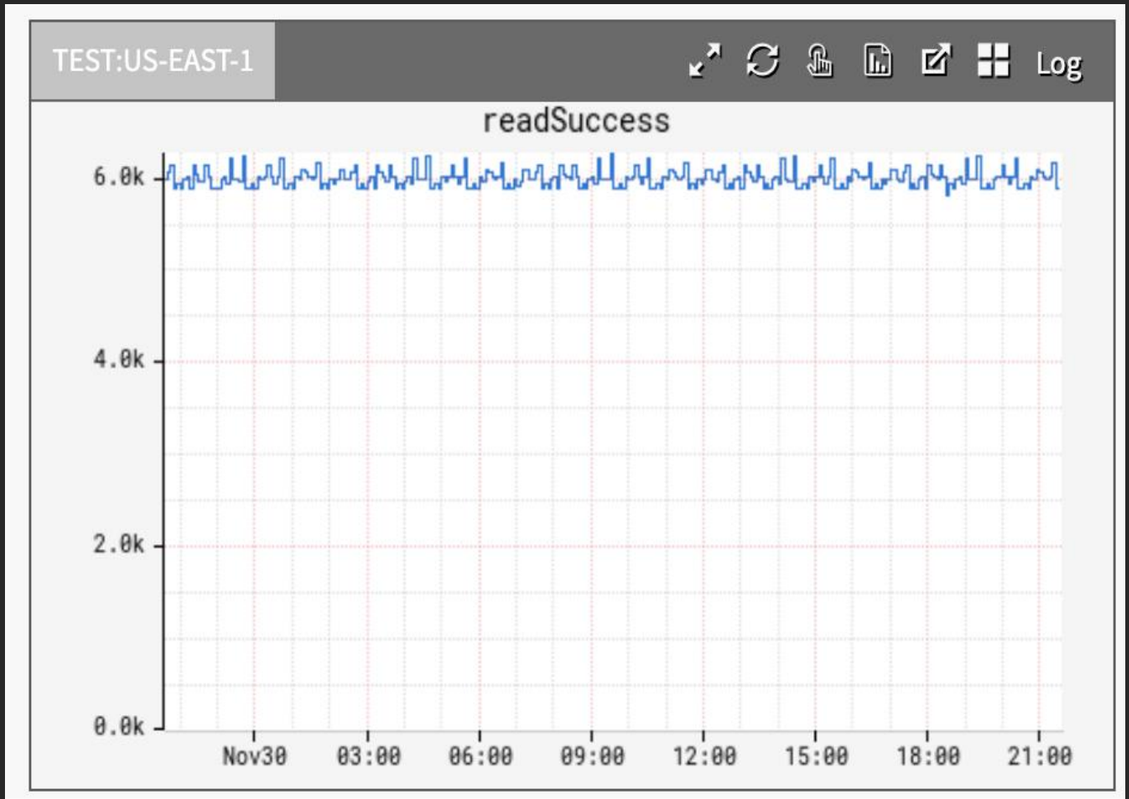
What to benchmark?

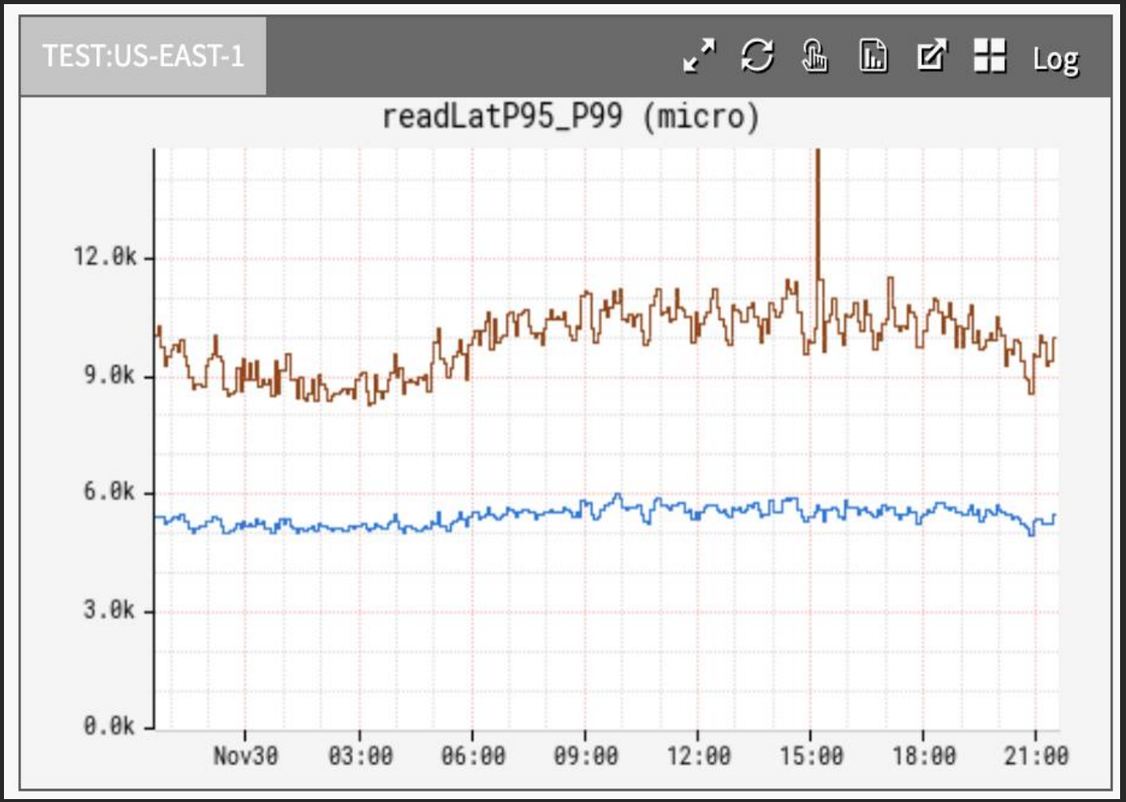
Use case to write to multiple tables as part of a transaction

- Test setup
  - 30 NDBench nodes reading and writing against DynamoDB tables
  - DynamoDB
    - One table – Key and 5 columns
    - 4 other tables – Key and 1 column
    - Column size – 1024 bytes

Results

Reads – 6000 rps at 10 milliseconds p99





Results

Writes – 1100 wps at 80 milliseconds p99





Lessons learned

~2% write failures observed

- Cause: DynamoDB tables being under-provisioned (*auto-scaling can be slow*)
- Fix: Pre-provision tables with the desired capacity

DynamoDB server metrics show capacity used for tables to be much lower than what is pre-provisioned

- Cause: Transaction writes consume double the *wcus*
- Fix: Just sit tight, RTM

Benchmarking Amazon S3

Maximizing available instance networking bandwidth



Benchmarking Amazon S3

We have seen Amazon S3 scale up to 50 Gbps on a single EC2 instance

Tune your clients/consumption patterns to get there …

https://gist.github.com/jolynch/aa53ca32f2eb7a72612ff0be6173e3d8

Lessons learned

- Bottleneck could be your usage pattern of a stateful service
- Analyzing benchmark results is a key for success

Benchmarking caching systems

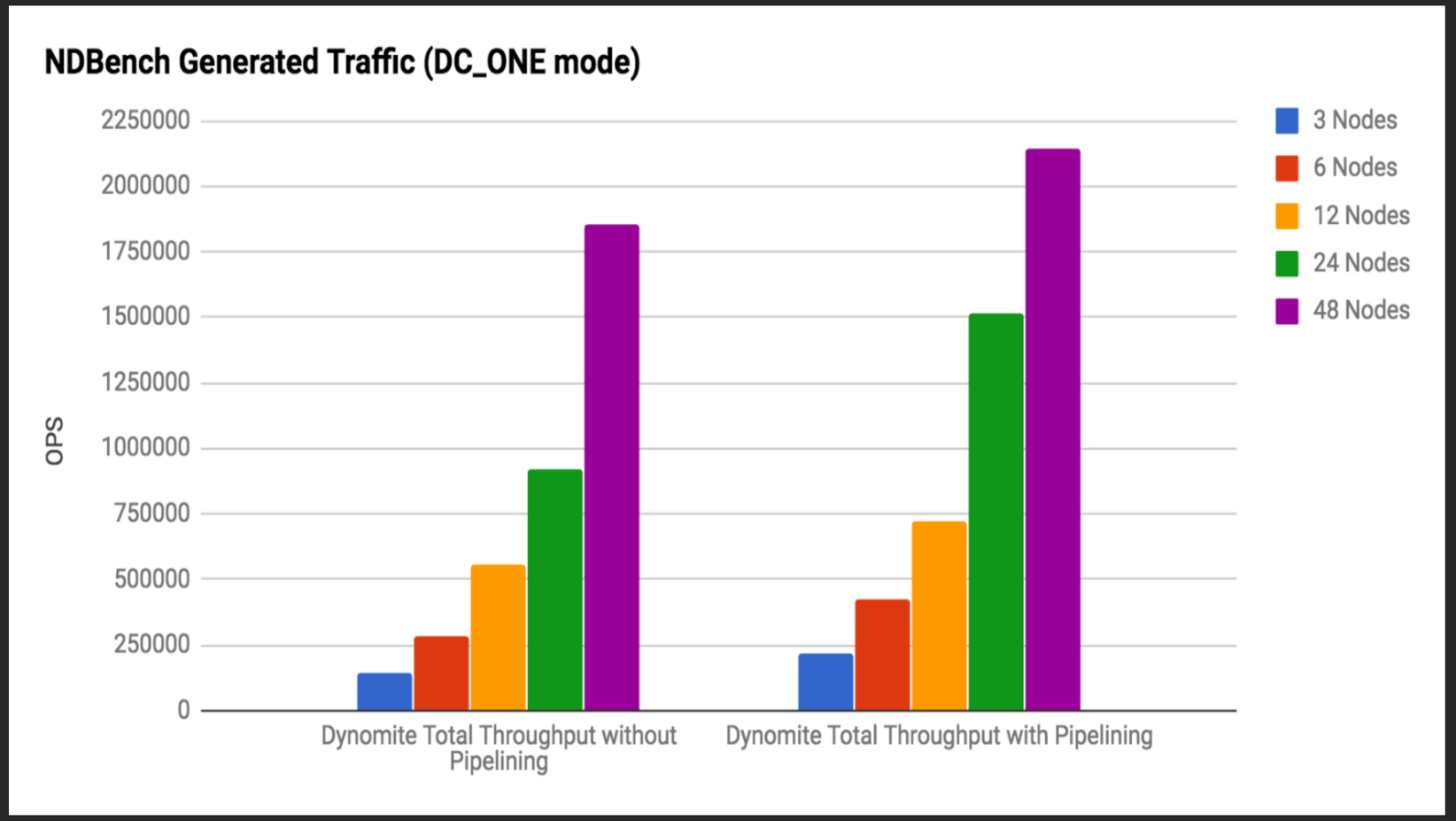
Focus heavily on performance

Goal

- Benchmark should not be the bottleneck
- Measuring the numbers with high precision

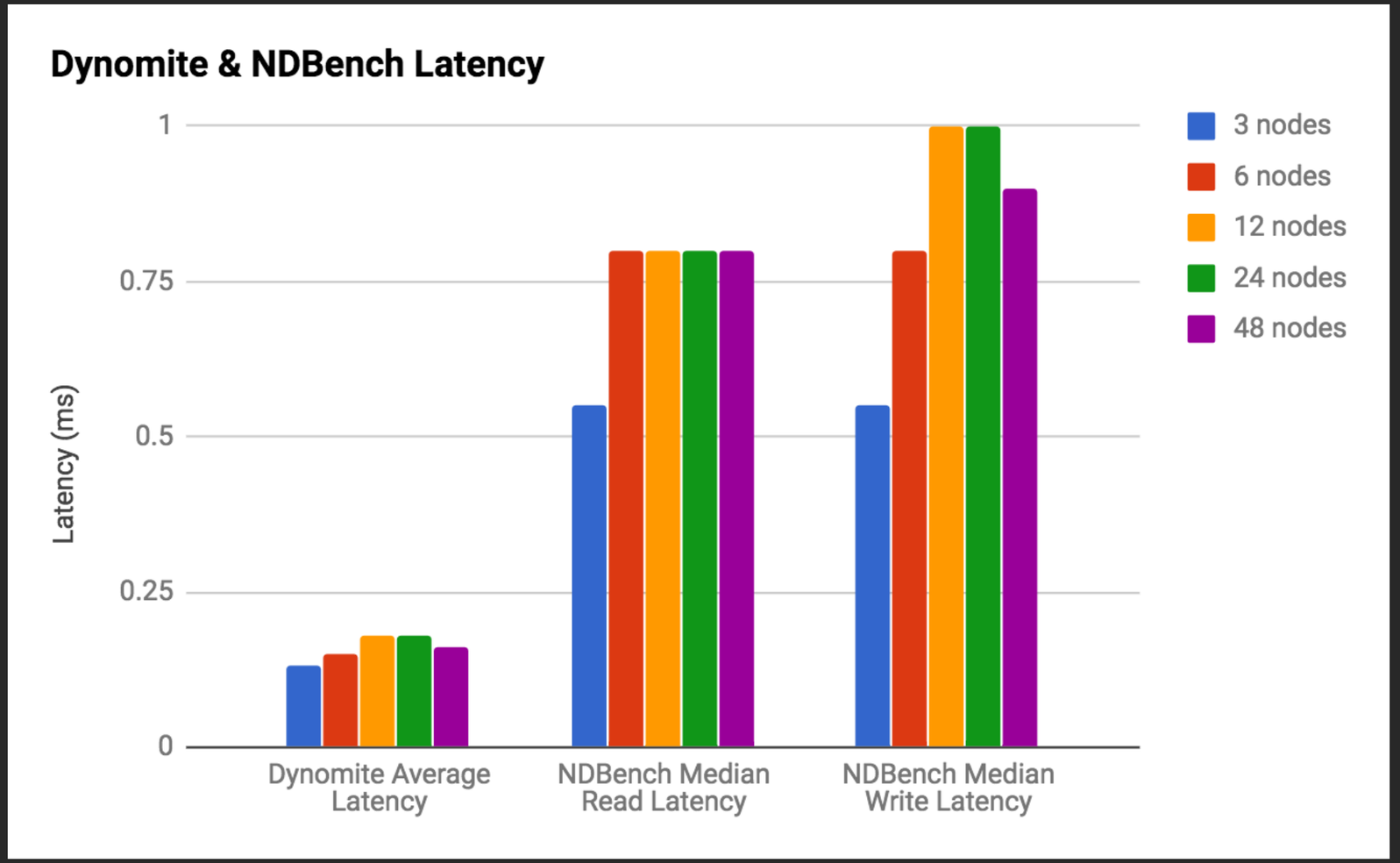
Throughput scalability





Latency


DYNOMITE


Dynomite & NDBench Latency
Latency (ms)
1
0.75
0.5
0.25
0
Dynomite Average Latency
NDBench Median Read Latency
NDBench Median Write Latency
3 nodes
6 nodes
12 nodes
24 nodes
48 nodes

Conclusions

- Let your benchmarking tool not be the bottleneck
- Measuring the result with great accuracy is important

Lessons learned

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Lessons learned

Both active and passive benchmarking are important

Use benchmarking to identify the bottlenecks

- AWS makes it easy to add the resources that you ran out of

Make it easy to benchmark

- Configuration-driven, auto detecting the pattern
- Self-driving benchmarks give better results than hand-tuned benchmarking

Lessons learned

Build a benchmarking repository like your test cases repository

- It is a continuous learning process; this job can never be complete

Invest in a desire-based control plane and push-based infrastructure

- Helps in easily building passive benchmarking systems

Lastly . . . benchmark your benchmarking tools

Takeaway

Benchmarking is a continuous development practice that should be integrated as part of your CI, CD, and runtime systems.

This cannot be forgotten as soon as your systems are deployed to production.

Thank you!

Vinay Chella

@vinaykchella

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.



Please complete the session survey in the mobile app.

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.