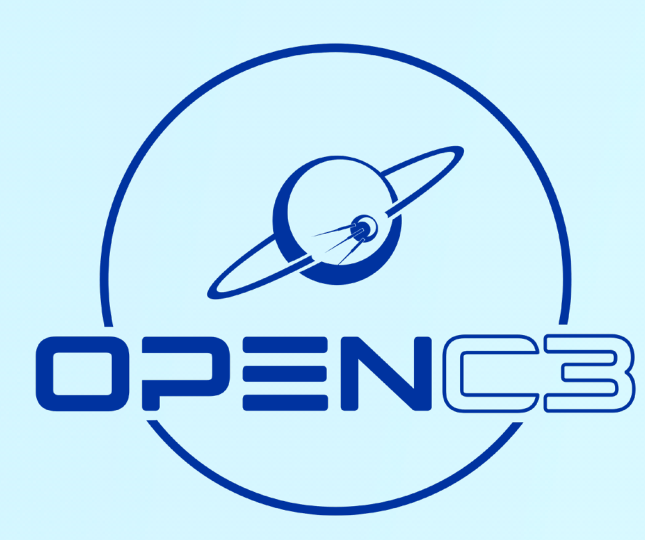
Scaling a C2 System to Hundreds of Satellites

Using Observability Technologies to Identify Performance Bottlenecks

Ryan Melton
Jason Thomas

© 2023 by OpenC3, Inc.
Published by The Aerospace Corporation with permission
Approved for Public Release

OpenC3 COSMOS Intro



Who, What, When, Where

- OpenC3 (Open Command, Control, and Communication) COSMOS is brought to you by OpenC3, Inc.
  - Founded by Ryan Melton and Jason Thomas - the authors of Ball Aerospace COSMOS
- COSMOS consists of a suite of applications to control embedded systems
- Over 17 years of heritage - Initial development in 2006, open sourced in 2014, re-architected in 2020, released independently as OpenC3 COSMOS in 2022.
- COSMOS is TRL-9 per NASA's Small Satellite State of the Art Report
- openc3.com and github.com/OpenC3/cosmos

Architecture!

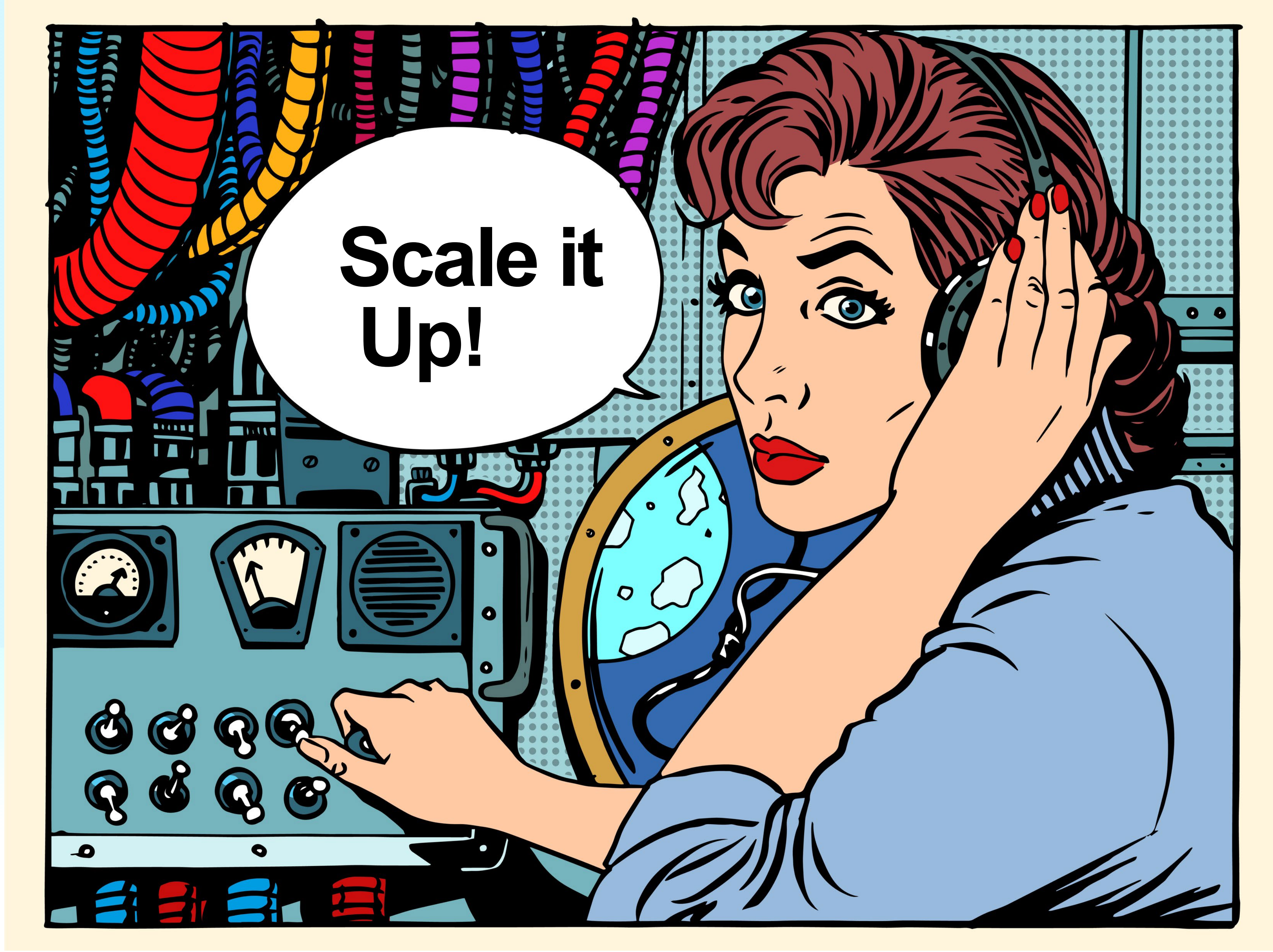
Telemetry Processing Architecture

Containerized Microservices Designed to Scale Horizontally

kubernetes	Kubernetes used for container orchestration across a cluster of nodes. For this experiment we used Google's GKE and Amazon's EKS.
docker	Telemetry Processing Chain Broken down into the following containers per target (satellite): Interface, Raw Packet Logging, Decommutation, Decom Packet Logging, Data Reduction
redis	Redis Cluster is the primary data store. Used as key/value store for configuration and current value table. Also used for pub/sub bus, and streaming message bus between containers.
amazon S3	Bucket Storage used for configuration, logged data, and as a static web server. For this experiment either Google Cloud Storage or Amazon S3.

Scalability Knobs

- Kubernetes Cluster:
  - Number of Nodes
  - vCPU/RAM/Disk per Node
  - Network Performance per Node
- Redis Cluster:
  - Number of Primary Nodes
  - Number of Replica Nodes
  - 16384 Hash Slots

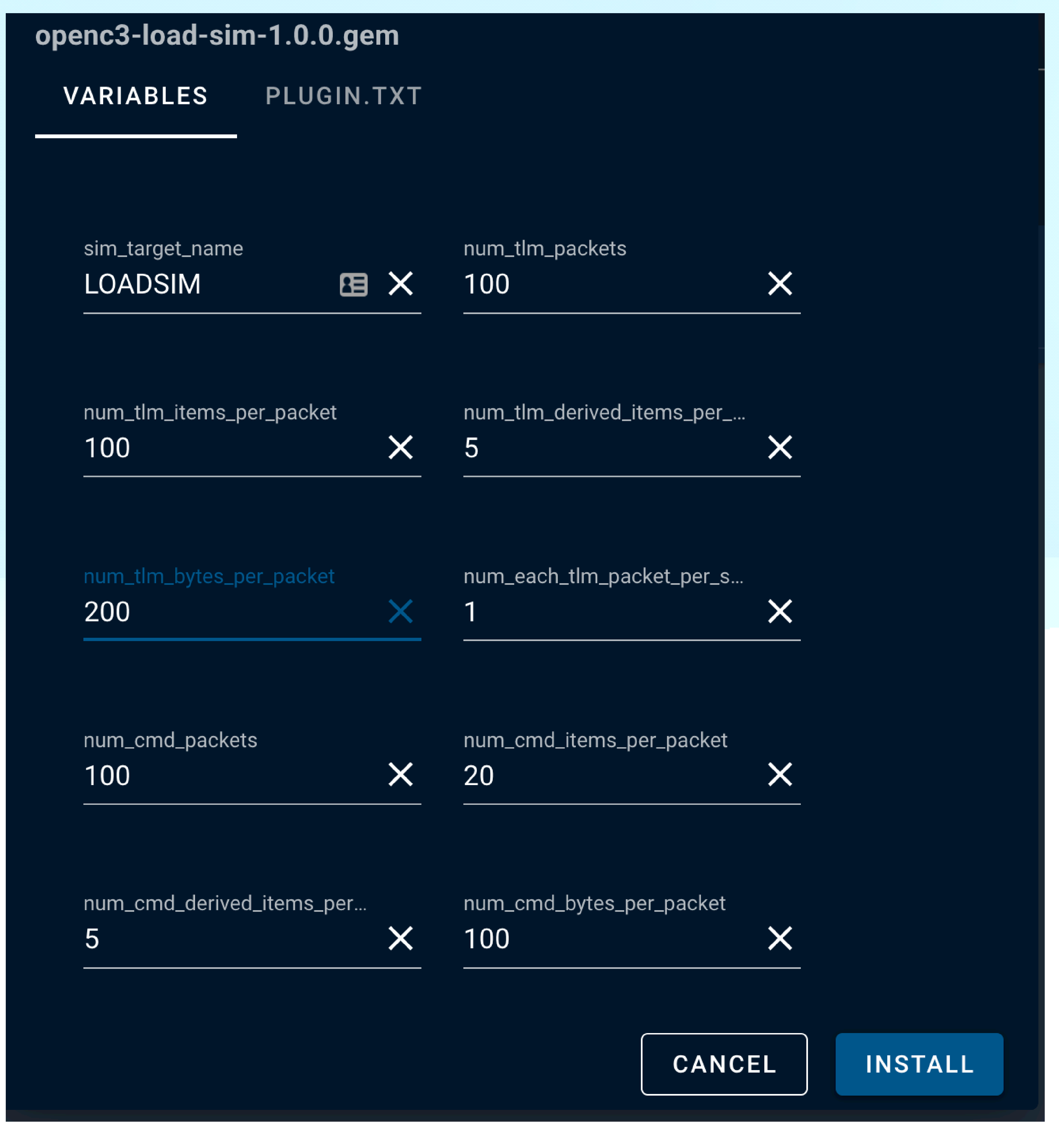


- COSMOS:
  - Microservice Independence
  - Packets Per Container

Satellite Load Simulation

Each Satellite Simulated with:

- 100 cmds / 100 tlm packets
- 25 items per cmd packet
- 105 items per tlm packet
- 100 Tlm Packets each 200 bytes at 1 Hz
- 100 * 200 = 20,000 bytes/sec = 160 Kbits/sec
- 10,500 Items Decommed Per Sec
- 200 Packets logged per second (Raw/Decom)



Open Source at: https://github.com/OpenC3/openc3-cosmos-load-sim

Let’s Run It!

Initial Test Setup

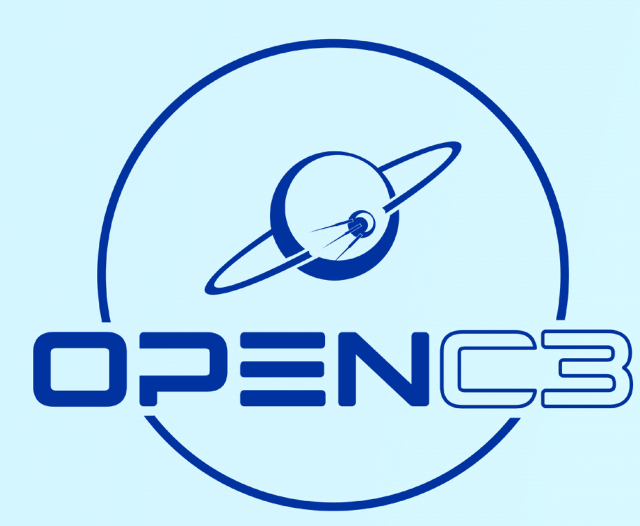
Google's GKE

- 4 Nodes @ e2-standard-4
- 4 vCPUs, 16GB of RAM
- x86
- Redis Cluster 3 Primary / 3 Replica
- Google Cloud Storage
- $0.54 per hour

Amazon EKS

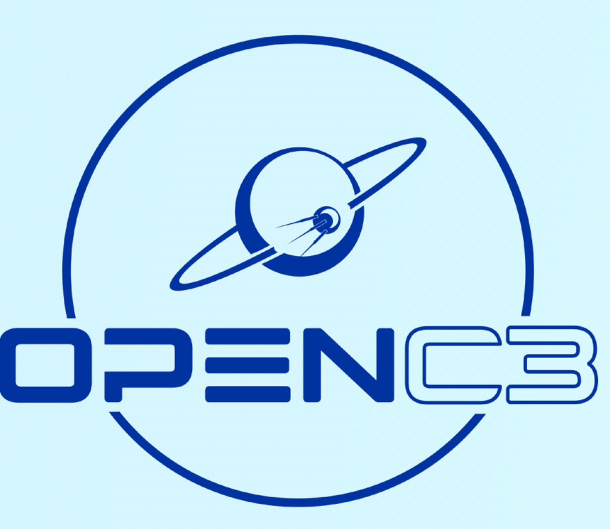
- 4 Nodes @ t4g.xlarge
- 4 vCPUs, 16GB of RAM
- Arm64 Graviton2, up to 5Gbps
- Redis Cluster 3 Primary / 3 Replica
- Amazon S3
- $0.54 per hour

Initial Test Results



- kubectl top nodes – Shows CPU and RAM utilization per node
  - Started showing an imbalance across the 4 nodes in our cluster
  - At 15 satellites – One node had reached 95% CPU utilization and decommutated data was delayed for targets mapped to that node
- Observed problems:
  - Our Kubernetes cluster was not evenly allocating CPU utilization across nodes based on actual usage
  - Our simulated targets are using about 60% vCPU each. Opportunity for optimization?
  - More internal metrics needed to let us know that things are starting to struggle

Initial Lessons Learned



- **Problem:** Kubernetes allocates containers to nodes based on the resource requests per container. If your containers don't request any CPU/Mem explicitly, then Kubernetes basically assumes they don't use any resources.
- **Response:** Need to at least guess at CPU/Mem requests for every container so that Kubernetes will spread them out across nodes evenly. Added a 100m / 100Mi resource request to each of our target containers.
- **Problem:** Difficult to detect when things start to fall behind
- **Response:** Added Prometheus support, and new metrics for key aspects of the telemetry processing pipeline. Most important new metrics:
    - Latency from data being placed on a stream to being read off
    - Decommutation time
    - Redis IOPS

Next and Final Test Setups

Amazon EKS

- 4 Nodes @ c7g.8xlarge
- 32 vCPUs, 64GB of RAM
- Arm64 Graviton3 / 15Gbps Network
- Redis Cluster 3 Primary / 3 Replica
- Amazon S3
- $4.64 per hour

Amazon EKS

- 4 Nodes @ c7g.8xlarge
- 32 vCPUs, 64GB of RAM
- Arm64 Graviton3 / 15Gbps Network
- Redis Cluster 15 Primary / 15 Replica
- Amazon S3
- $4.64 per hour

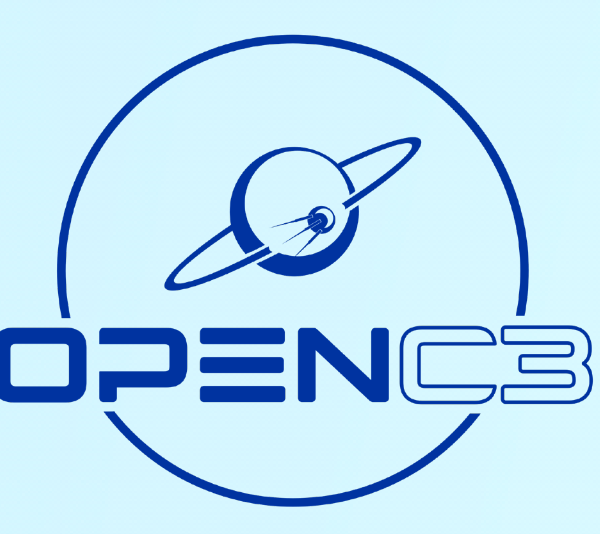
Next and Final Test Results

- Somewhere between 50 and 75 satellites, the 3 node Redis cluster started to be overwhelmed.
- Redis connect errors showing up in logs, and delayed processing
- Increased the 3 node Redis cluster to 15 nodes for final test.
- Successfully scaled up to 150 satellites with everything running smoothly
- At 160 satellites was still functioning but showing some variability
- Somewhere between 160 and 170 satellites became unable to keep up
- Noticed uneven allocation of satellites across Redis cluster nodes

```
openc3-redis-ephemeral-0     ●  1/1   0 Running   214   956    42   n/a   191
openc3-redis-ephemeral-1     ●  1/1   0 Running   222   706    44   n/a   141
openc3-redis-ephemeral-2     ●  1/1   0 Running   608  1512   121   n/a   302
openc3-redis-ephemeral-3     ●  1/1   0 Running   468  1501    93   n/a   300
openc3-redis-ephemeral-4     ●  1/1   0 Running   302  1031    60   n/a   206
openc3-redis-ephemeral-5     ●  1/1   0 Running   255   942    51   n/a   188
openc3-redis-ephemeral-6     ●  1/1   0 Running   435  1632    87   n/a   326
openc3-redis-ephemeral-7     ●  1/1   0 Running   433  1288    86   n/a   257
openc3-redis-ephemeral-8     ●  1/1   0 Running   316  1212    63   n/a   242
openc3-redis-ephemeral-9     ●  1/1   0 Running   369  1249    73   n/a   249
openc3-redis-ephemeral-10    ●  1/1   0 Running   412  1530    82   n/a   306
openc3-redis-ephemeral-11    ●  1/1   0 Running   375  1437    75   n/a   287
openc3-redis-ephemeral-12    ●  1/1   0 Running   326   958    65   n/a   191
openc3-redis-ephemeral-13    ●  1/1   0 Running   377  1329    75   n/a   265
openc3-redis-ephemeral-14    ●  1/1   0 Running   508  1907   101   n/a   381
```

Metrics Scaling up to 170 Satellites

Lessons Learned and Areas for Future Improvement



- Redis cluster sharding assigns each satellite to a random Redis cluster node.
- Random assignment eventually leads to unbalanced shards
- Need to add Redis cluster balancing / scaling functionality to our Admin interface in future versions
- AWS initially limited our account to 32 vCPUs. Had to request an increased limit to run these tests. Granted in about 2 hours.
- AWS limits each node to 234 pods/containers. Scaling beyond 200 satellites would require adding more Kubernetes nodes

INTERFACES | TARGETS | CMD PACKETS | TLM PACKETS | ROUTERS | STATUS

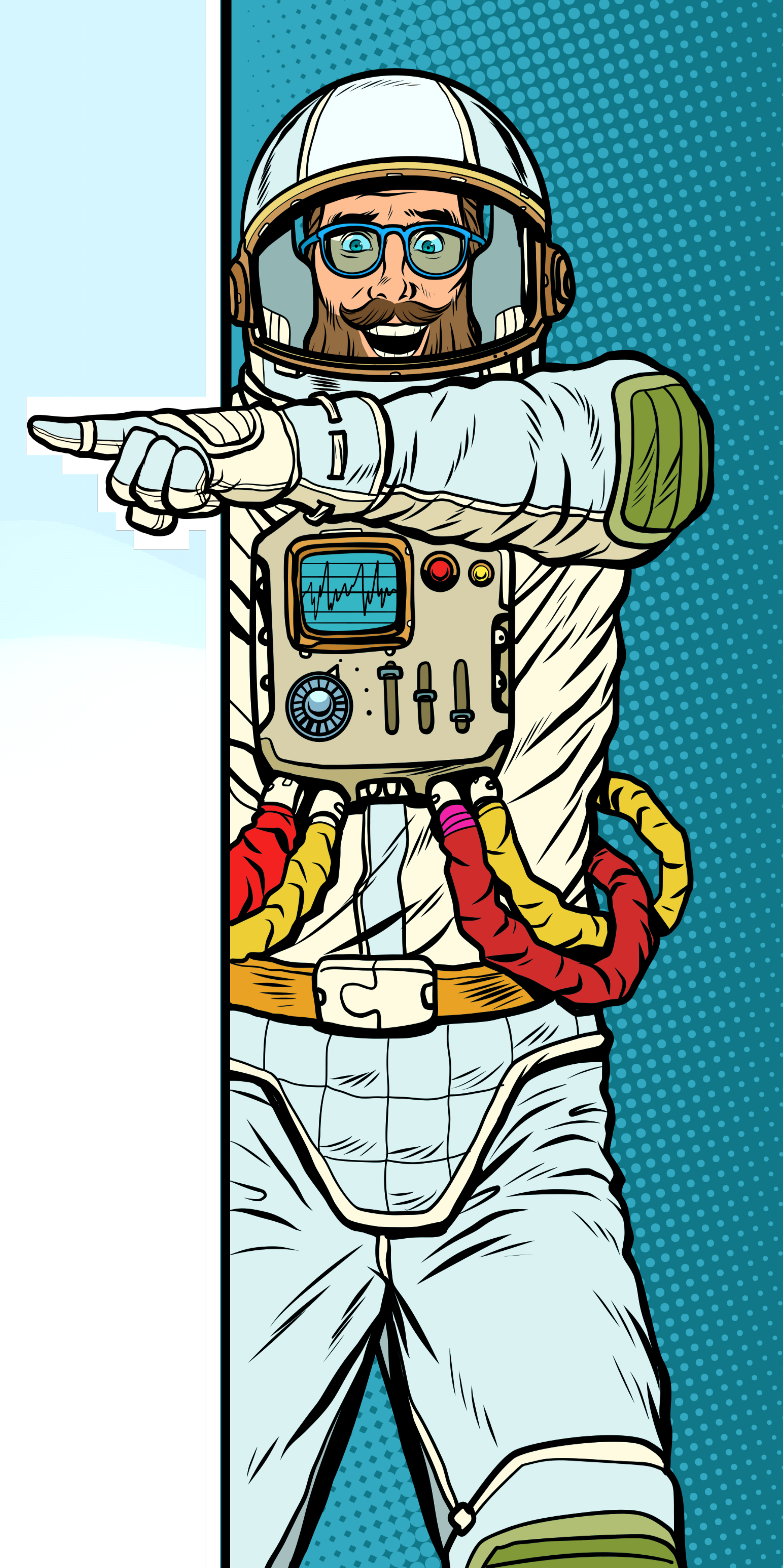
160 Interfaces

Search

Name	Connect / Disconnect	Connected	Clients	Tx Q Size	Rx Q Size	Tx Bytes	Rx Bytes	Cmd Pkts	Tlm Pkts
INT_LOADSIM	DISCONNECT	CONNECTED	0	0	0	0	109696115	0	535103
INT_LOADSIM10	DISCONNECT	CONNECTED	0	0	0	0	89605500	0	437100
INT_LOADSIM100	DISCONNECT	CONNECTED	0	0	0	0	52794265	0	257533

Summary and Conclusions

- The OpenC3 COSMOS architecture can successfully scale to support 100s of satellites
  - Demonstrated up to 160 satellites
- Having the right metrics really matters when scaling
- Demonstrated:
  - 16,000 telemetry packets per second
  - 1,680,000 telemetry points per second
  - 3,200,000 Mbytes/sec, 25,600,000 Mbits/sec



QUESTIONS

?

For more info, check us out at opencc3.com