RESILIENCE
REALIZED

KubeCon | CloudNativeCon
North America 2021

KubeCon | CloudNativeCon
North America 2021

RESILIENCE
REALIZED

Argo And Tekton:

Pushing The Boundaries Of The Possible On Kubernetes

Alex Collins (Intuit) and Jason Hall (Red Hat)

Who are we?

Alex Collins

- Intuit
- Kubernetes and OLTP
- Lead Engineer on Argo
- Likes coffee and cycling

Jason Hall

- Red Hat
- Devtools for ~8 years
- Co-founded Tekton
- Likes pizza and sitting

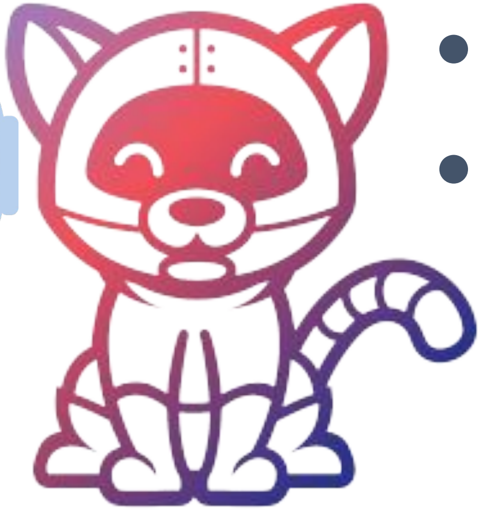
What do we do?

Argo Workflows

- General-purpose workflows
- Steps run sequentially
- Tasks run in a DAG
- Built on Kubernetes
- Cute logo

Tekton

- CD-focused workflows
- Steps run sequentially
- Tasks run in a DAG
- Build on Kubernetes
- Cute logo



Why build on Kubernetes?

- Node management
- Workload scheduling
- Custom resources: flexible, extensible API (w/ "free" RBAC)
- 🥰 ***Community!*** 🥰
    - security
    - performance
    - observability
    - portability
    - client tooling
    - multi-tenancy
    - policy enforcement
    - ...and tons more!

However...

- Kubernetes was designed for running *apps*
  - Deployments, Services, Ingress, etc.
- Assumes long-running pods → we need short-running pods
- Assumes long-running containers → short-running containers
- No control over container lifecycle → we need to start/stop processes
- No convention for communication between containers or pods → we need to pass data around efficiently

For Example

- Container Lifecycle (start/stop, sidecars).
- Container IPC.
- Cross-Pod communication.
- Custom Resource Proliferation.

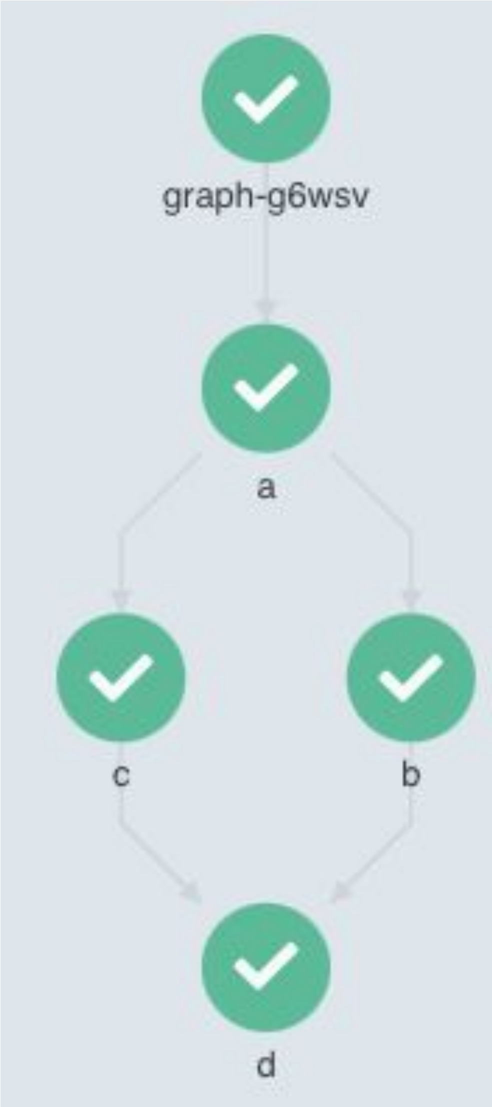
Container Lifecycle

Container Lifecycle

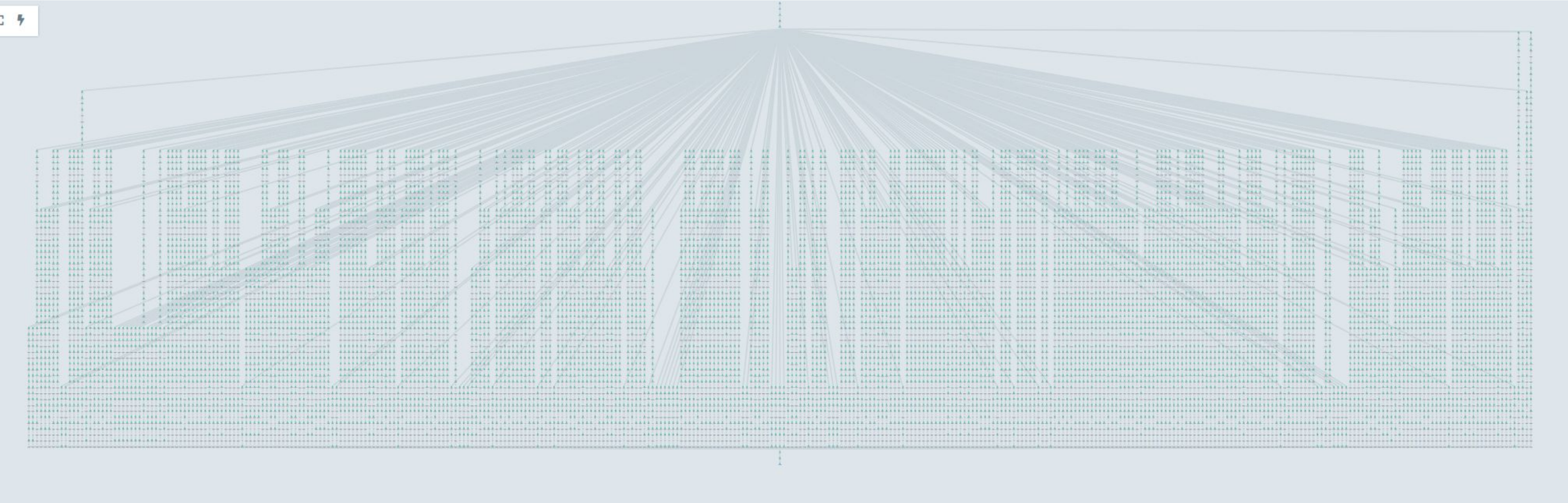
What we'd like:

- Execute processes in a graph
- Graph maybe dynamic,

  can change at runtime.



Container Lifecycle



† It’s not unusual to see a workflow running 20k concurrent pods - but that’s another talk entirely...

Ordering Container Start-Up

Ordering options out of the box:

- Init containers.
- One container per pod.

Ordering Container Shutdown

What we'd like:

- Shutdown containers in a controlled order.
- Containers must be allowed to complete work on shutdown.
- Must work with pod deletion (SIGTERM).

“Commandlet” Pattern aka Entrypoint Rewriting aka Emissary Executor

Entrypoint Rewriting

- Binary is mounted on an shared volume.
- New **entrypoint command** that runs the original command as a sub-process.

```
apiVersion: v1
kind: Pod
spec:
 volumes:
   - name: var
     emptyDir: { }
 initContainers:
   - name: init
     volumeMounts:
       - mountPath: /var/run/app
         name: var
     command: [ cp ]
     args: [ entrypoint, /var/run/app/entrypoint ]
 containers:
   - name: main
     volumeMounts:
       - mountPath: /var/run/app
         name: var
     command: [ /var/run/app/entrypoint ]
     args: [ original_command... ]
```

Ordering Container Start-Up

- Wait for some condition before starting the sub-process.
- Signal the sub-process.
- Capture the sub-processes’ stdout, exit code, etc.
- Wait for some condition before exiting.
- Directly access the container file system.

Ordering Container Start-Up

But…

- Requires the container to be running while waiting to start sub-process

  → resource requests can be set to mitigate cost
- Cannot dynamically add containers to the graph.

Shutdown with SIGTERM

Ordering Container Shutdown

Containers should gracefully exit on SIGTERM

How to send SIGTERM?

- Delete the pod
- Use `kubectl exec -- kill 1`

Ordering Container Shutdown

`kubectl exec -c main -- /bin/kill 1` does not work if:

- No kill binary: Debian (builtin), scratch, or distroless.
- If you’re running a shell script.
- If you’re running as PID != 1 (e.g. as non-root).

Ordering Container Shutdown

Mitigations:

- Use “dumb-init” to handle SIGTERM.
- Like entrypoint, mount /var/run/app/kill.
- Use $(pidof command)

Entrypoint Rewriting

Previously...

KubeCon | CloudNativeCon

North America 2019

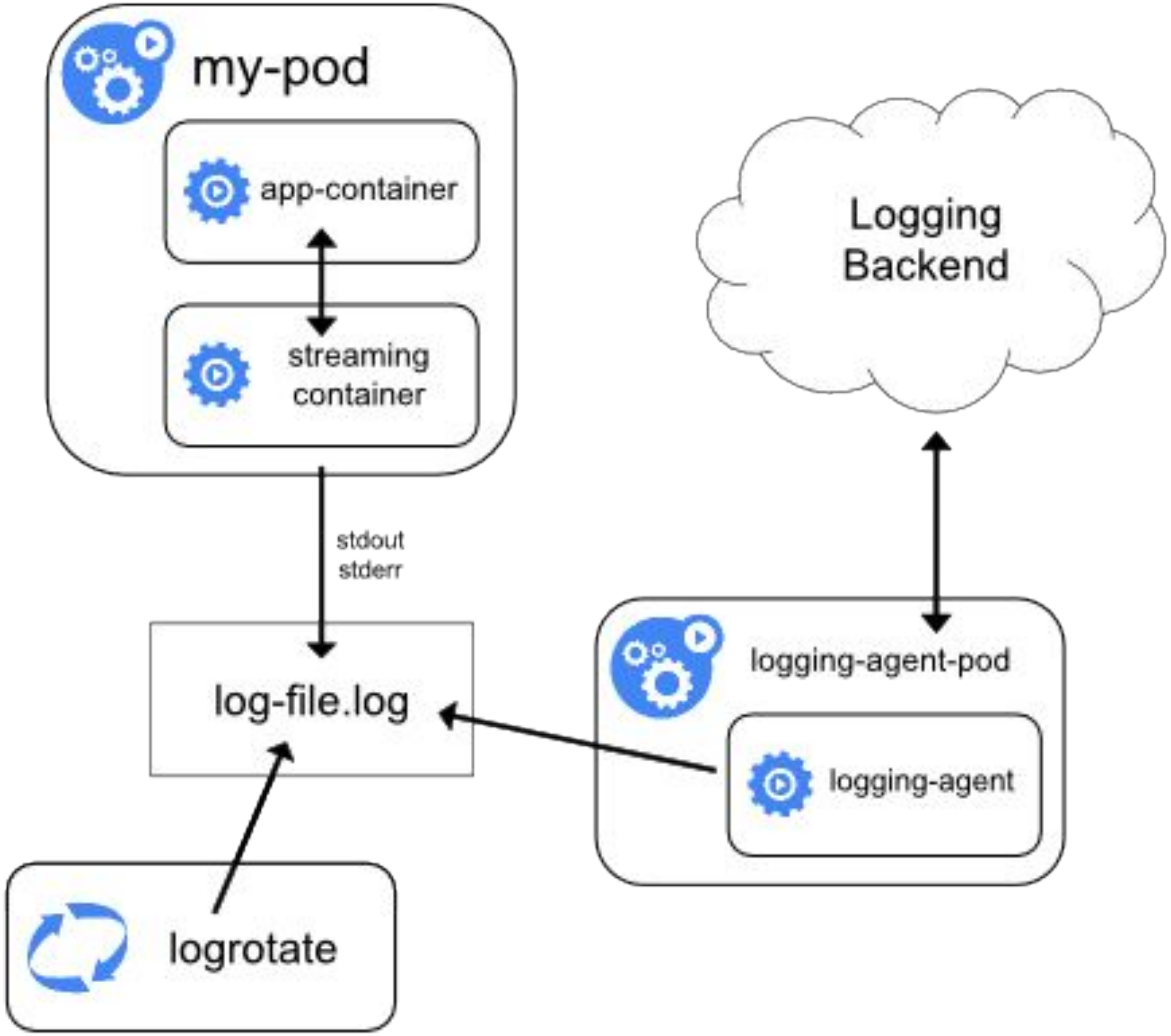
Russian Doll: Extending Containers with Nested Processes

Christie Wilson & Jason Hall, Google

https://www.youtube.com/watch?v=iz9_omZ0ctk

Sidecars

Sidecars


my-pod
app-container
streaming container
stdout
stderr
log-file.log
logrotate
Logging Backend
logging-agent-pod
logging-agent

Sidecars

- We don’t control:
  - ○ The APIs they expose,
  - ○ how they behave,
  - ○ or if they have an init on them.

Inject Sidecars: Istio or Vault

- We don’t even know about **injected sidecars:**

  so no entrypoint re-writing, and no shared empty dir.
- “/quitquitquit” is not a standard.

Today’s solution:

- Disable Istio or Vault :(

RESILIENCE REALIZED

KubeCon | CloudNativeCon
North America 2021

Container Inter-Process Communication (C-IPC)

Container IPC

What we’d like to do between containers:

- Share data/files.
- Remote procedure call (RPC).
- Streaming data.

Container IPC is just *nix IPC

Shared files

Memory mapped files

Shared memory

Semaphores

Pipes (named and unnamed)

Message queues (POSIX/Sys-V)

Sockets

Signals

Container IPC

Call the **Kubernetes API**:

- Limited in what you can do, e.g. delete/log/exec.
- Needs escalated permissions.
- Network requests exit the host.

Container IPC

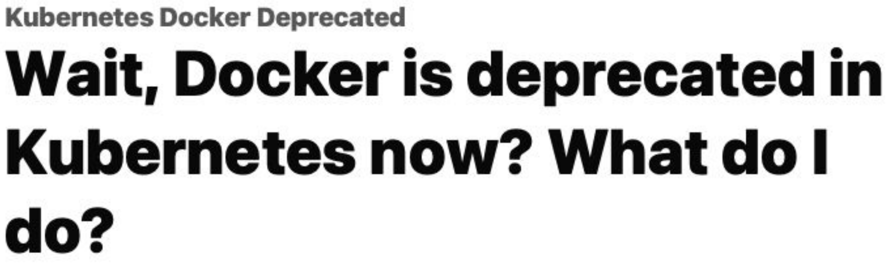
Call the **Kubelet API** :)

- Network requests stay on localhost.
- God mode.
- Esoteric and not well understood.

Container IPC - Docker API

Call the **Docker API**:

- Basically `docker cp` and `docker kill`
- God mode.
- Deprecated.

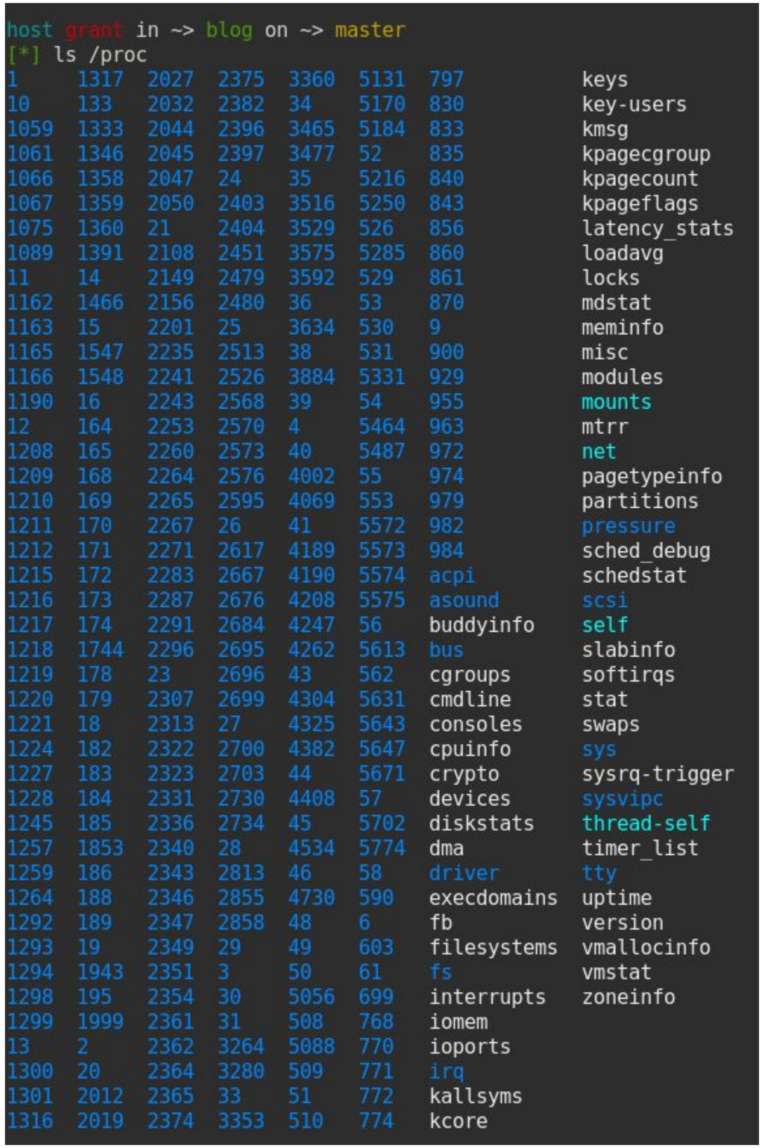

Kubernetes Docker Deprecated

Wait, Docker is deprecated in Kubernetes now? What do I do?

Container IPC - PNS

Use **Process Namespace Sharing (PNS):**

- Mount procfs (i.e. /proc).
- Access to both processes and file handles.
- User mode - PID != 1.
- Does not work well with:
  - Run-as-non-root (UNIX file permissions).
  - Short-lived process (<10s).
  - Complex.



```
host grant in ~> blog on ~> master
[*] ls /proc
1     1317  2027  2375  3360  5131  797          keys
10    133   2032  2382  34    5170  830          key-users
1059  1333  2044  2396  3465  5184  833          kmsg
1061  1346  2045  2397  3477  52    835          kpagecgroup
1066  1358  2047  24    35    5216  840          kpagecount
1067  1359  2050  2403  3516  5250  843          kpageflags
1075  1360  21    2404  3529  526   856          latency_stats
1089  1391  2108  2451  3575  5285  860          loadavg
11    14    2149  2479  3592  529   861          locks
1162  1466  2156  2480  36    53    870          mdstat
1163  15    2201  25    3634  530   9            meminfo
1165  1547  2235  2513  38    531   900          misc
1166  1548  2241  2526  3884  5331  929          modules
1190  16    2243  2568  39    54    955          mounts
12    164   2253  2570  4     5464  963          mtrr
1208  165   2260  2573  40    5487  972          net
1209  168   2264  2576  4002  55    974          pagetypeinfo
1210  169   2265  2595  4069  553   979          partitions
1211  170   2267  26    41    5572  982          pressure
1212  171   2271  2617  4189  5573  984          sched_debug
1215  172   2283  2667  4190  5574  acpi         schedstat
1216  173   2287  2676  4208  5575  asound       scsi
1217  174   2291  2684  4247  56    buddyinfo    self
1218  1744  2296  2695  4262  5613  bus          slabinfo
1219  178   23    2696  43    562   cgroups      softirqs
1220  179   2307  2699  4304  5631  cmdline      stat
1221  18    2313  27    4325  5643  consoles     swaps
1224  182   2322  2700  4382  5647  cpuinfo      sys
1227  183   2323  2703  44    5671  crypto       sysrq-trigger
1228  184   2331  2730  4408  57    devices      sysvipc
1245  185   2336  2734  45    5702  diskstats    thread-self
1257  1853  2340  28    4534  5774  dma          timer_list
1259  186   2343  2813  46    58    driver       tty
1264  188   2346  2855  4730  590   execdomains  uptime
1292  189   2347  2858  48    6     fb           version
1293  19    2349  29    49    603   filesystems  vmallocinfo
1294  1943  2351  3     50    61    fs           vmstat
1298  195   2354  30    5056  699   interrupts   zoneinfo
1299  1999  2361  31    508   768   iomem
13    2     2362  3264  5088  770   ioports
1300  20    2364  3280  509   771   irq
1301  2012  2365  33    51    772   kallsyms
1316  2019  2374  3353  510   774   kcore
```

Container IPC

Use **shared empty-dir volumes**:

- Communicate using marker files (e.g. /var/run/app/data).
- Process polls for file changes (slow).
- We think you can use FIFOs too (fast, but unproven).
- Simple, secure, and robust.

Great for “slow IPC”.

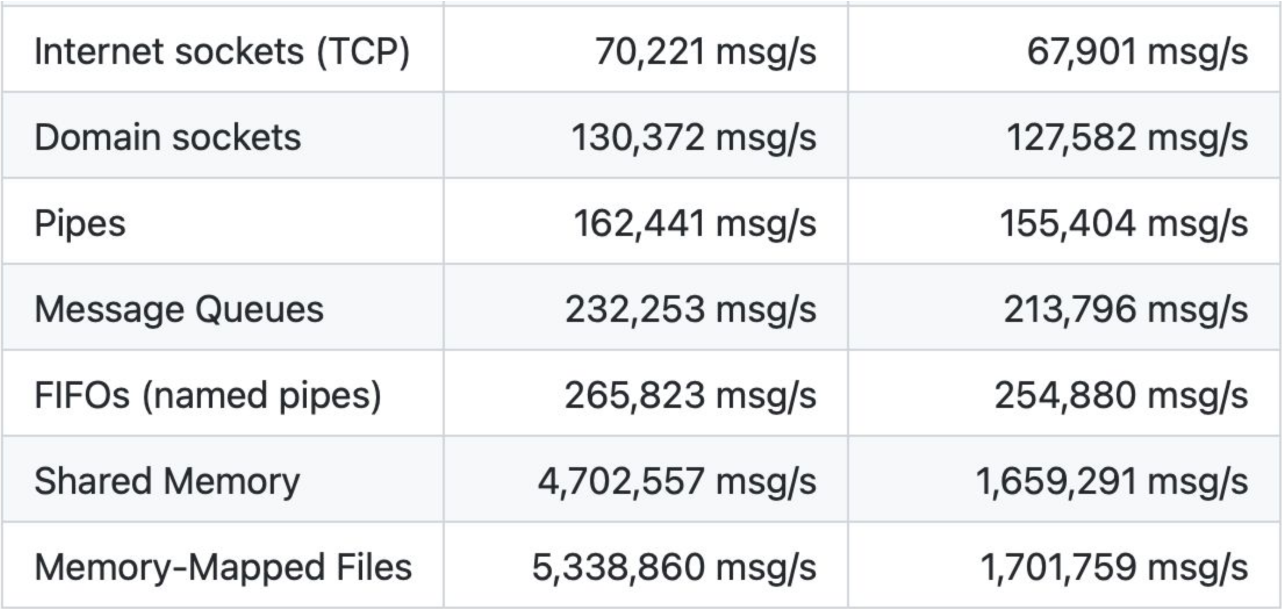
Container IPC

Use **HTTP**:

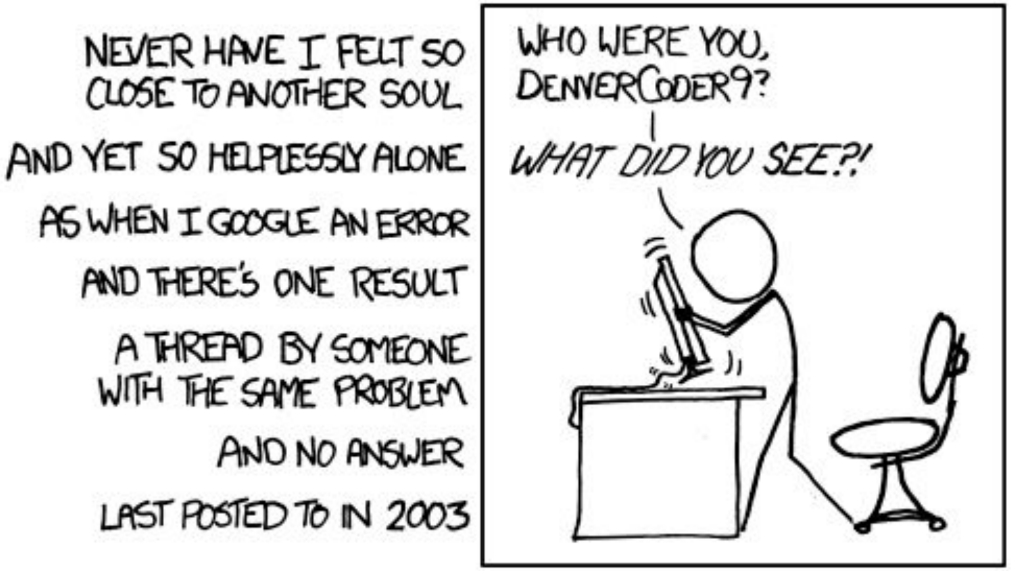
- Well-known and easy to implement.
- You must define an API contact.
- Secure (own network namespace).
- Fast (when using HTTP keep-alives or Unix Domain Sockets).

Great for “fast IPC”.

Other ways?



Internet sockets (TCP)	70,221 msg/s	67,901 msg/s
Domain sockets	130,372 msg/s	127,582 msg/s
Pipes	162,441 msg/s	155,404 msg/s
Message Queues	232,253 msg/s	213,796 msg/s
FIFOs (named pipes)	265,823 msg/s	254,880 msg/s
Shared Memory	4,702,557 msg/s	1,659,291 msg/s
Memory-Mapped Files	5,338,860 msg/s	1,701,759 msg/s



NEVER HAVE I FELT SO CLOSE TO ANOTHER SOUL
AND YET SO HELPLESSLY ALONE
AS WHEN I GOOGLE AN ERROR
AND THERE'S ONE RESULT
A THREAD BY SOMEONE WITH THE SAME PROBLEM
AND NO ANSWER
LAST POSTED TO IN 2003
WHO WERE YOU, DENVERCODER9?
WHAT DID YOU SEE?!

Cross-Pod Communication

Cross-Pod Communication

- Need to pass data between Tasks (Pods)
  - e.g., built container image digest, fetched git commit SHA
  - One Task's `results` is another Task's `inputs`
- Need to expose results to users through Kubernetes API
- Short-lived containers, that we don't control
  - No HTTP communication built in

terminationMessage

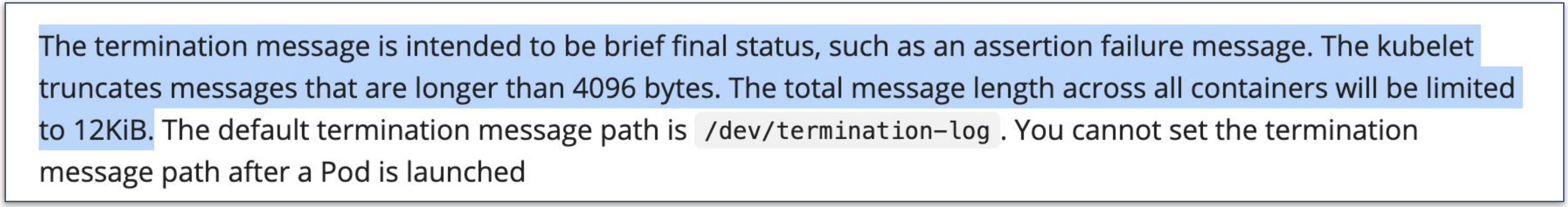
- Your container, when it exits, can write data that kubelet collects
  - `.status.containerStatuses[].terminationMessage`
  - `/dev/termination-log` by default
  - Configurable with `.spec.containers[].terminationMessagePath`

`terminationMessage`

- TaskRun Pod writes to `/tekton/results/blah`
- Injected entrypoint sees that and writes to the Pod's `terminationMessagePath` as JSON
- Tekton controller sees it in the Pod, extracts it to TaskRun status
- Passes TaskRun results to other TaskRun inputs at startup
- Also use this to report step start times, etc.

terminationMessage

- Limits:



The termination message is intended to be brief final status, such as an assertion failure message. The kubelet truncates messages that are longer than 4096 bytes. The total message length across all containers will be limited to 12KiB. The default termination message path is `/dev/termination-log`. You cannot set the termination message path after a Pod is launched

Write to ConfigMap / other CRD

- Write to a temporary "scratchpad" ConfigMap or custom resource
- Read it from the controller, then delete it
- Scope RBAC down as narrowly as possible
- That's what Argo does!

Write to ConfigMap / other CRD

- Disadvantages:
  - Define and maintain this type
  - API server load: create and delete objects, define RBAC for them

Custom Resource Proliferation

Custom Resource Proliferation

- CRDs are *great*
  - ...but they're not magic.
- It's all **etcd** in the end
  - Bytes stored
  - Objects stored
  - Update QPS
- Destabilizing etcd is *bad*

Mitigations

- Don't use Jobs when you only need Pods
- Avoid unnecessary updates
    - Batch updates at the end of a reconcile loop
- Avoid duplicating info across multiple objects (QPS)
- Avoid monolithic mega-objects (QPS, size)
- Avoid lots of little objects (QPS, # of objects)
- Offloading into another database
    - with a pointer in your resource

Mitigations

- Enforce `ResourceQuota`
- Prune old resources
  - By age?
  - By # of resources?
  - Lose history :(
- Tekton Results / Argo Workflow Archive
  - Copy execution results to separate DB, then prune ~immediately
  - Better indexing and search
  - Needs CLI + UI integration

What can we do?

What can we do?

- Do nothing; keep hacking around it

✨KEPs!✨

KEPs!

- Container start/stop API?
  - or declare container DAG dependencies?
- Standardize commandlet pattern?
- ResourceQuota improvements
  - `deleteOldest:true?`

Standardize Commandlet Pattern

- Standard HTTP and `kubectl exec` APIs
- Tried and tested
- Security?

Container DAG API (sketch)

```
spec:
  containers:
  - name: step-1
  - name: step-2
    waitForTermination: ['step-1']
  - name: step-3
    waitForTermination: ['step-2']
    timeout: 30s
```

Container DAG API (sketch)

```
spec:
  containers:
  - name: step-1
  - name: step-2
    status: ~~Stopped~~
  - name: step-3
    status: Killed!
```

Containers sub-resource (sketch)

Start a container:

```
POST /api/v1/namespace/my-ns/pods/my-pod/containers -d {‘name:
‘main’,...}
```

```
PATCH /api/v1/namespace/my-ns/pods/my-pod -d {‘spec’:
‘containers’: [{‘name’: ‘main’,...}]}
```

Send a signal to the root process:

```
POST /api/v1/namespace/my-ns/pods/my-pod/containers/main/signal
-d {‘signal’: ‘SIGTERM’}
```

Just some ideas.
What about injected sidecars?

Custom Resource pruning

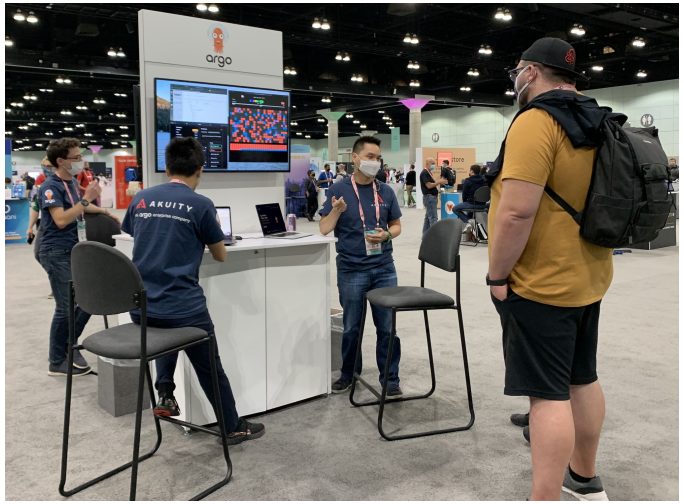
- Add to `ResourceQuota`?

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: prune-taskruns
  namespace: my-ns
spec:
  hard:
    count/taskruns.tekton.dev: 100
  deleteOldest: true
  # TODO: limit to finished TaskRuns
```

Questions!

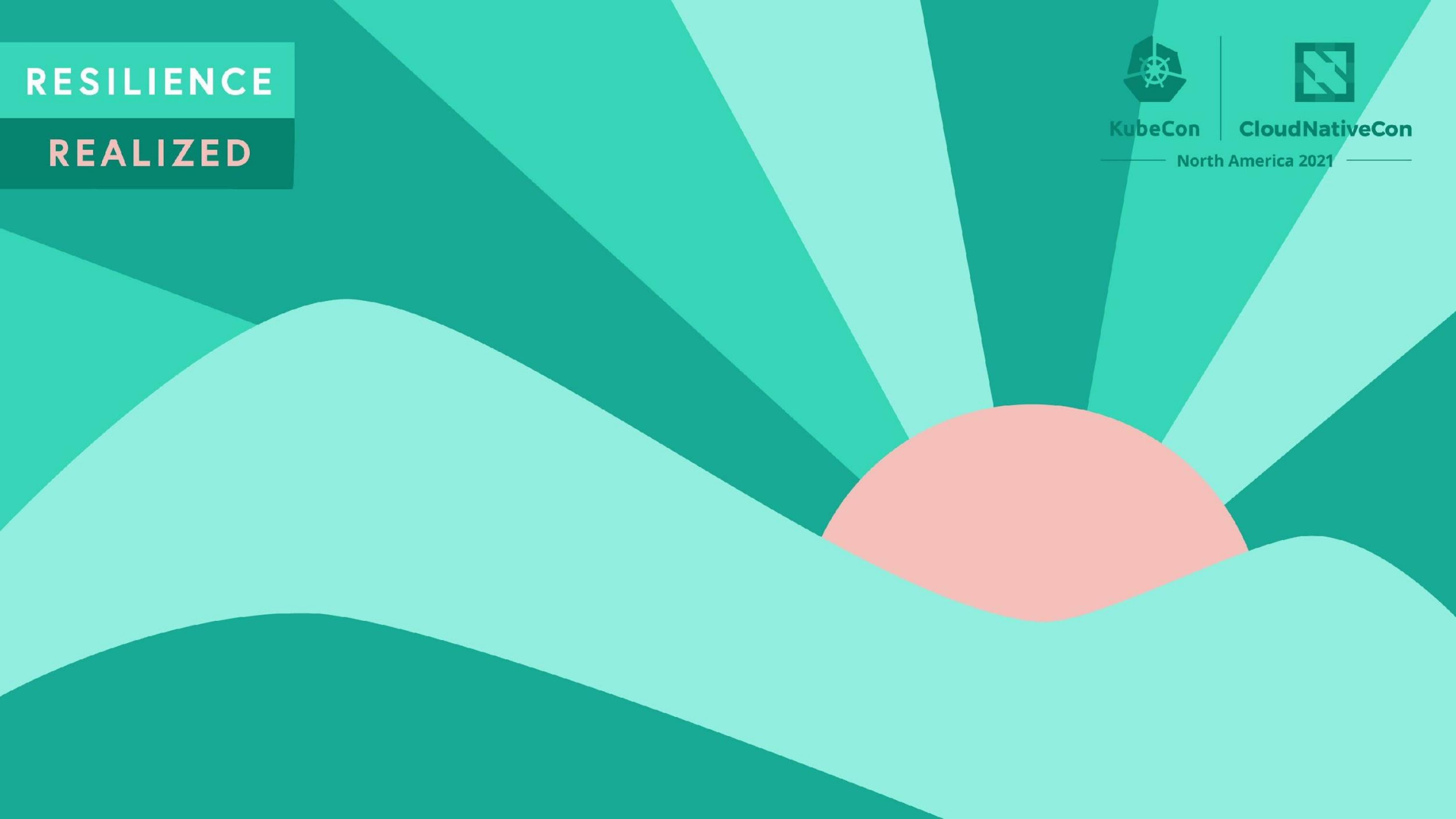
Find out more...

- The Intuit booth (Zone Sage).
- RedHat booth.
- Argo booth in CNCF Pavilion.



RESILIENCE
REALIZED

KubeCon | CloudNativeCon
North America 2021



zzz

TITLE

- Z
- Z
- Z
- Z
- Z
- Z
- Z