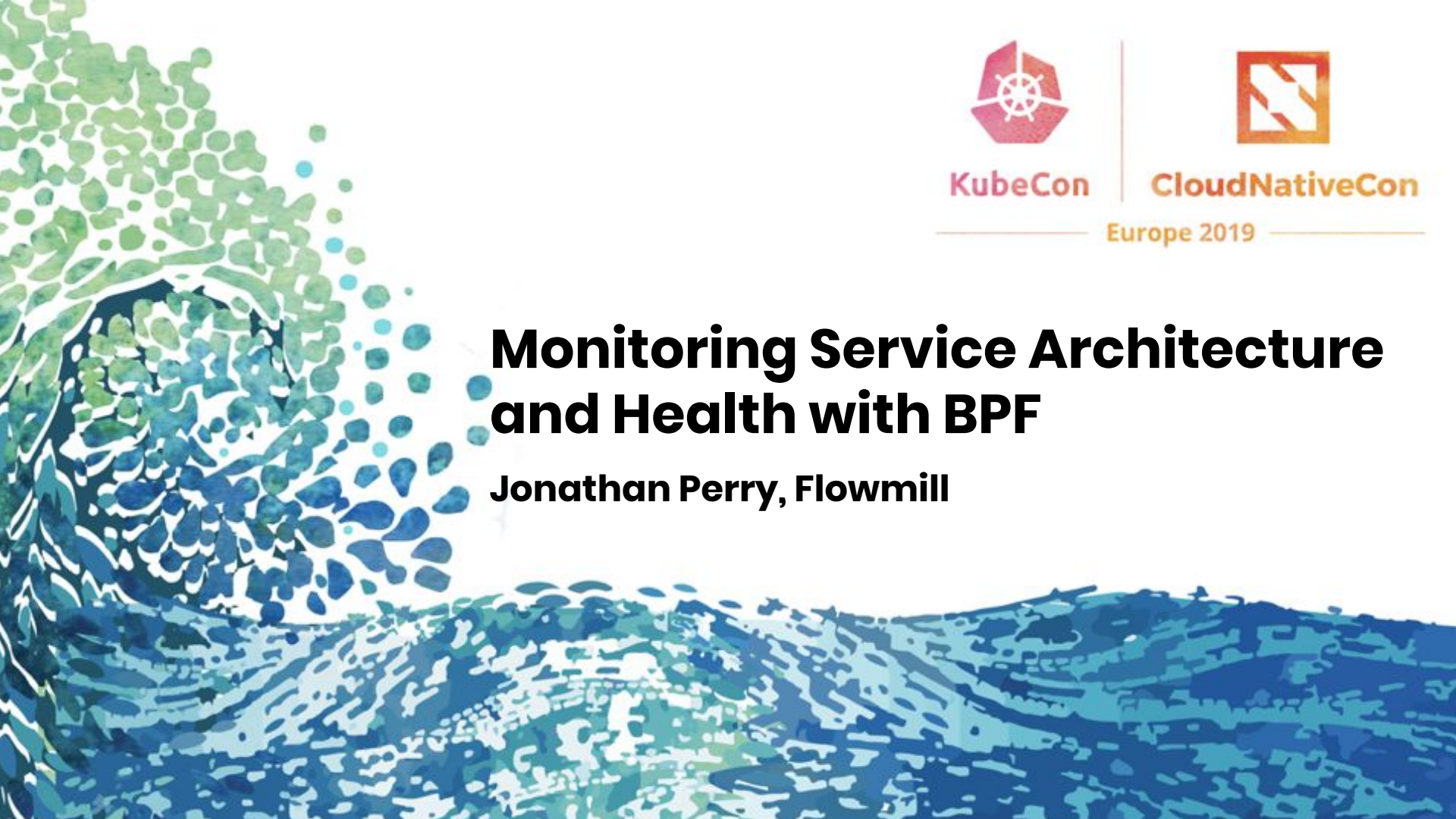


Monitoring Service Architecture and Health with BPF

Jonathan Perry, Flowmill

Agenda

- **Demo**: visibility into **Architecture, Health and Cost**
- How that worked: **Linux + Kubernetes - "Flow monitoring"**
- Flow vs App monitoring: **Pros and Cons**
- Major challenges: **Performance & coverage → use eBPF**
- Building a complete system: **Collection & analysis architecture**
- Is all this really practical?: **Evaluation**
- Where next: **Adding Application monitoring** (and how)

Hi! I'm Jonathan Perry

jperry@flowmill.com

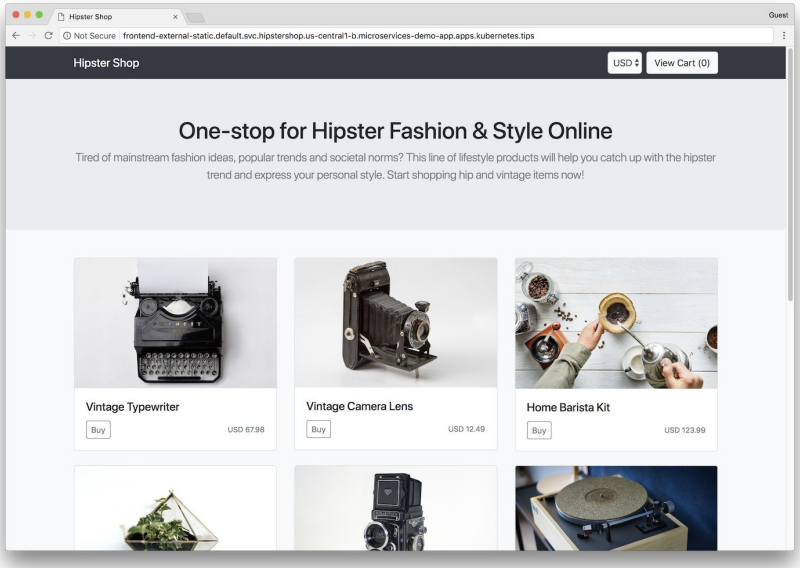
www.flowmill.com

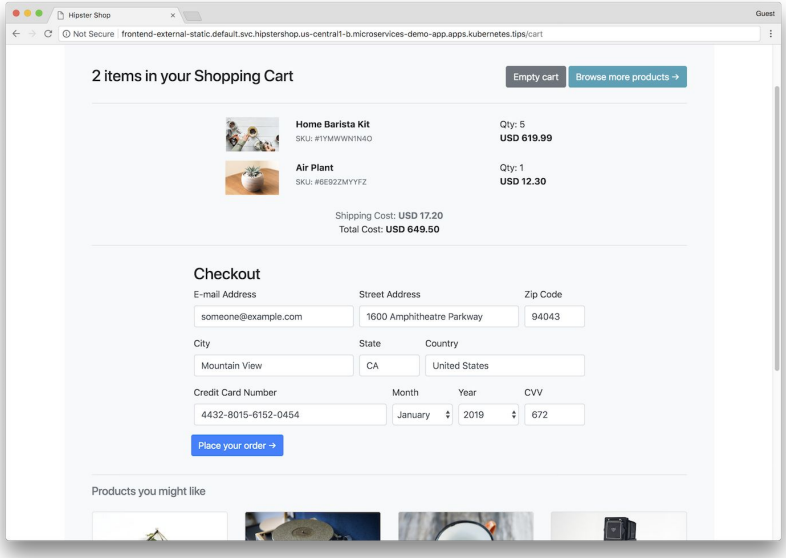
- Government: large-scale deployments
- MIT PhD: extreme monitoring systems
  - prod at Facebook
- Flowmill: CEO

Demo application

GoogleCloudPlatform / microservices-demo

Watch 144 Unstar 4,698 Fork 596





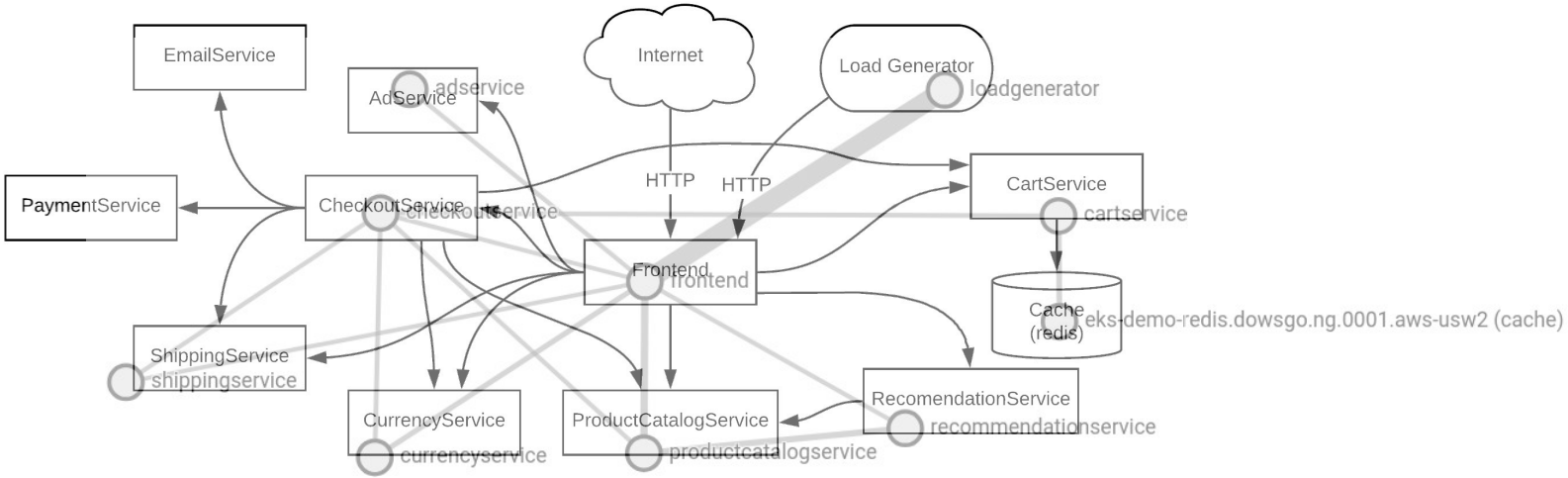
github.com/GoogleCloudPlatform/microservices-demo

Visibility #1: Architecture

```
NAME                                     READY   STATUS    RESTARTS   AGE
adservice-6cd6965787-lmjrl               1/1     Running   0          1d
cartservice-75f55fbc45-dv85s             1/1     Running   22         1d
checkoutservice-6848667dd7-jt44p         1/1     Running   0          1d
currencyservice-668f49f985-l42s6         1/1     Running   0          1d
emailservice-796bb9588b-cfqcp            1/1     Running   0          1d
frontend-6dcd4969b4-v2vqq                1/1     Running   0          14h
loadgenerator-54d77df7b-tgnwc            1/1     Running   0          1d
paymentservice-548657568f-p5hb7          1/1     Running   0          1d
productcatalogservice-7b94dfb45c-9gz7k   1/1     Running   0          1d
recommendationservice-5fb85f46df-hwj5s   1/1     Running   0          1d
shippingservice-5f5d75bf65-v5p7f         1/1     Running   0          1d
```

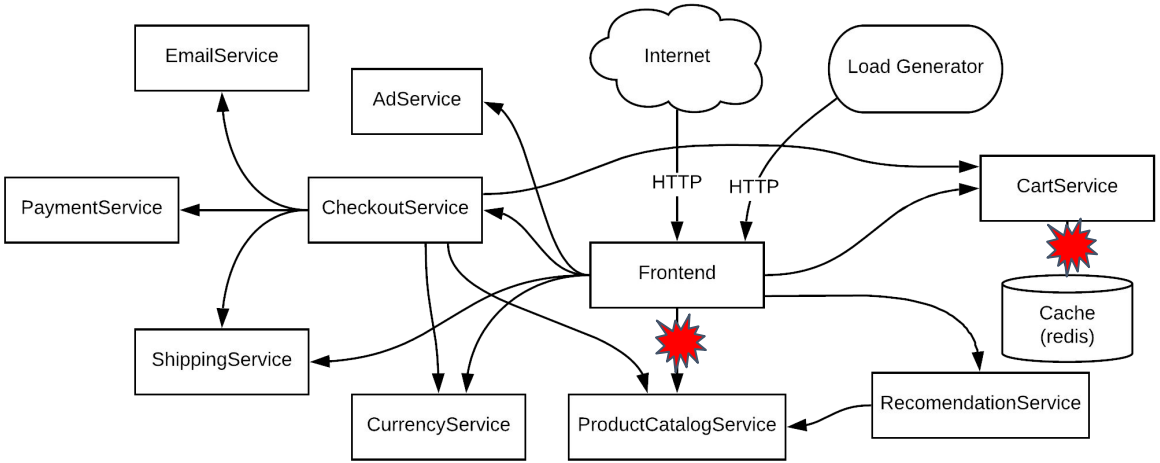
this shows the components, but how do they interact?

Visibility #1: Architecture



Just deployed, are old dependencies ok? New dependency?
Are Dev and prod isolated?
HA: what zones are communicating?

Visibility #2: Health

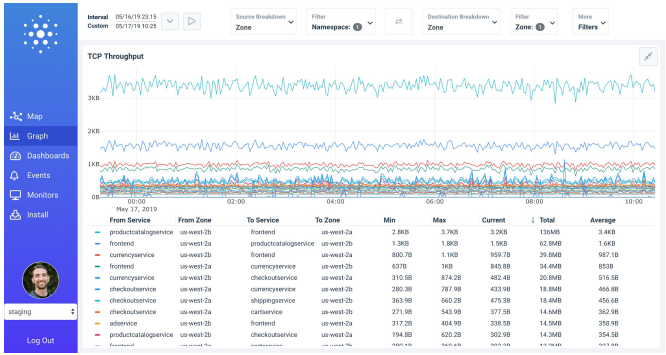


- Demo: Detecting service degradation
- Demo: Detecting security group misconfiguration

Visibility #3: Cost

Top service bandwidth consumption

- per Node
- across Zones → Demo
- across Regions



From Service	From Zone	To Service	To Zone	Min	Max	Current	Total	Average
productcatalogservice	us-west-2b	frontend	us-west-2a	2.8KB	3.7KB	3.2KB	136MB	3.4KB
frontend	us-west-2a	productcatalogservice	us-west-2b	1.3KB	1.8KB	1.5KB	62.8MB	1.6KB
currencyservice	us-west-2b	frontend	us-west-2a	800.7B	1.1KB	959.7B	39.8MB	987.1B
frontend	us-west-2a	currencyservice	us-west-2b	637B	1KB	845.8B	34.4MB	853B
currencyservice	us-west-2b	checkoutservice	us-west-2a	310.5B	874.2B	482.4B	20.8MB	516.5B
checkoutservice	us-west-2a	currencyservice	us-west-2b	280.3B	787.9B	433.9B	18.8MB	466.8B
checkoutservice	us-west-2a	shippingservice	us-west-2b	363.9B	660.2B	475.3B	18.4MB	456.6B
checkoutservice	us-west-2a	cartservice	us-west-2b	271.9B	543.9B	377.5B	14.6MB	362.9B
adservice	us-west-2b	frontend	us-west-2a	317.2B	404.9B	338.5B	14.5MB	358.9B
productcatalogservice	us-west-2b	checkoutservice	us-west-2a	194.8B	620.2B	302.9B	14.3MB	354.5B

So far...

- Architecture
  - real-time map (when deploying)
  - high availability architecture
- Health
  - service degradation
  - security group / firewall
- Cost
  - across zones

→ with Flow Data

How: Flow data

Linux:

Timestamp	Source	Destination	Ports	Bytes	Drops	RTT
1418530010	172.31.16.139	172.31.16.21	20641 22	4249	2	4 ms

K8s:

IP	Pod	Image	Tag	Zone
172.31.16.139	frontend	frontend-image	v1.16	us-west-1c
172.31.16.21	checkoutservice	checkout-image	v2.12a	us-west-1a

Joined:

Timestamp	Source	Destination	Ports	Bytes	Drops	RTT
1418530010	frontend frontend-image v1.16 us-west-1c	checkout checkout-image v2.12a us-west-1a	20641 22	4249	2	4 ms

Flow monitoring: Pros and cons

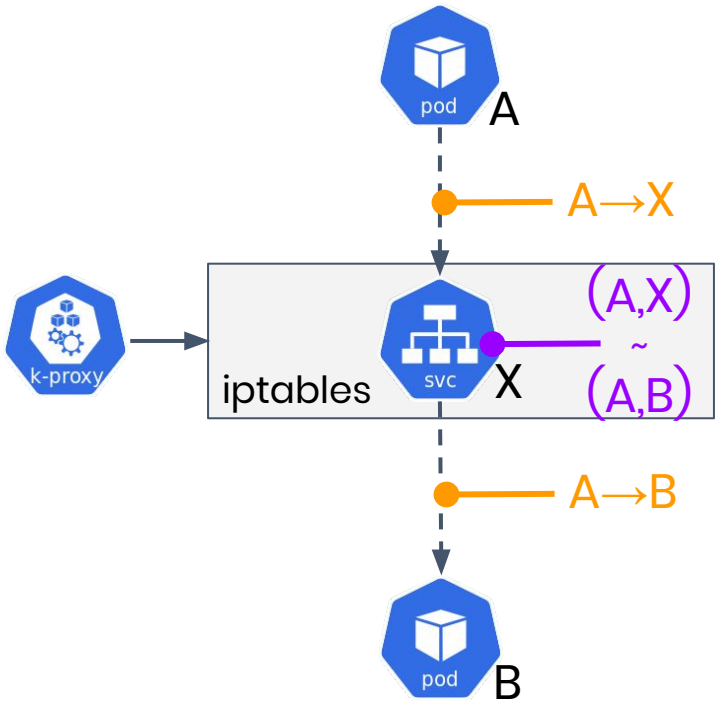
Pros:

- **No code changes** - only use info from Linux+k8s
- **100% coverage** - same reason
- **Small overhead** - few, optimizable collection points
  - more on this in “Evaluation” section
- **External visibility** - observe managed services, APIs

Cons:

- **No application-level error codes**
  - only see proxies (throughput, rtt, drops)
  - solvable - more towards end of talk

Getting Flow Data




```
$ kubectl describe pod $POD

Name:           A
Namespace:      staging
...
Status:         Running
IP:             100.101.198.137
Controlled By:  ReplicaSet/A
```

```
# PID=`docker inspect -f '{{.State.Pid}}' $CONTAINER` \
   nsenter -t $PID -n ss -ti

ESTAB  0   0    100.101.198.137:34940   100.65.61.118:8000
       cubic wscale:9,9 rto:204 rtt:0.003/0 mss:1448 cwnd:19
ssthresh:19 bytes_acked:2525112 segs_out:15664 segs_in:15578
data_segs_out:15662 send 73365.3Mbps lastsnd:384
lastrcv:10265960 lastack:384 rcv_space:29200 minrtt:0.002
```

A X

```
# conntrack -L

tcp      6 86399 ESTABLISHED src=100.101.198.137
dst=100.65.61.118 sport=34940 dport=8000
src=100.101.198.147 dst=100.101.198.137 sport=8000
dport=34940 [ASSURED] mark=0 use=1
```

A

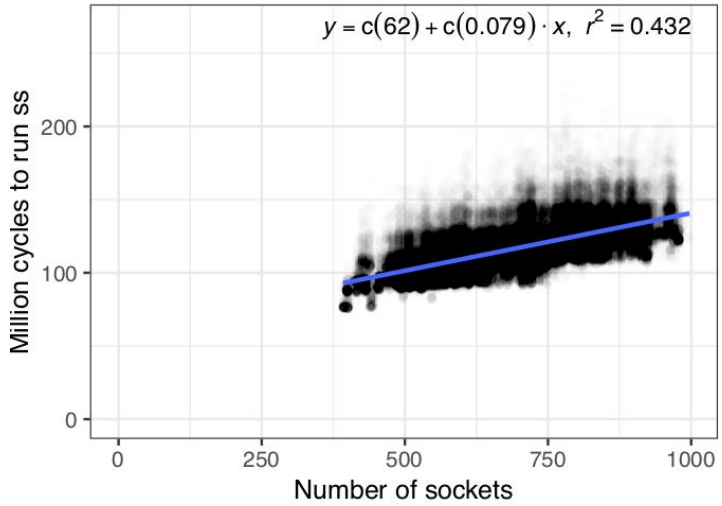
X

B

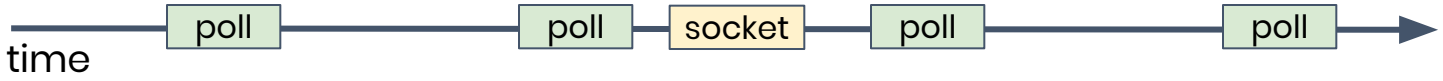
A

CLI performance & coverage

- **Performance**:
  - iterates over all sockets
  - built for CLI use (printfs)



- **Coverage**: Linux CLI tools are polling based



→ Misses events between polls

Enter eBPF

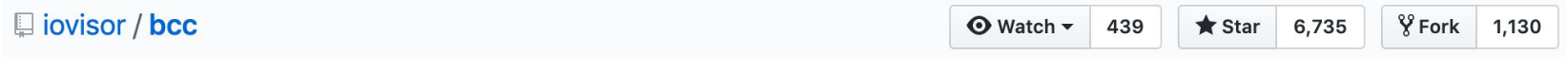
- Linux bpf() system call since 3.18
- Run code on kernel events
- Only changes, more data

- Safe: In-kernel verifier, read-only
- Fast: JIT-compiled

Unofficial BPF mascot by Deirdré Straughan

→ **100% coverage + no app changes + low overhead ftw!**

Using eBPF



Demo:

to run a bcc container:

```
docker run -it --rm \
  --privileged \
  -v /lib/modules:/lib/modules:ro \
  -v /usr/src:/usr/src:ro \
  -v /etc/localtime:/etc/localtime:ro \
  --workdir /usr/share/bcc/tools \
  --pid=host \
  zlim/bcc
```

https://github.com/iovisor/bcc/blob/master/QUICKSTART.md
+ host pid namespace

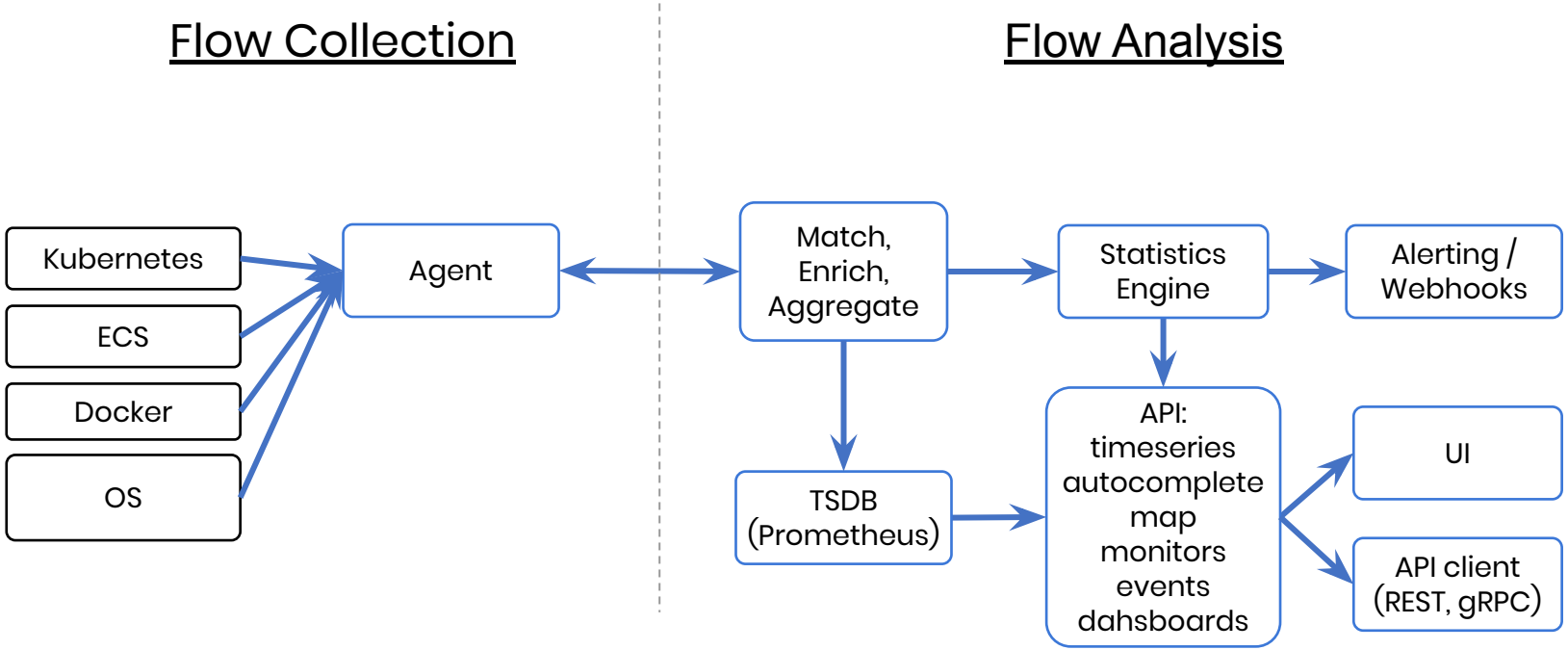
tcptop:

- instruments tcp_sendmsg and tcp_cleanup_rbuf
- need to be careful of races:

```
# IPv4: build dict of all seen keys
ipv4_throughput = defaultdict(lambda: [0, 0])
for k, v in ipv4_send_bytes.items():
    key = get_ipv4_session_key(k)
    ipv4_throughput[key][0] = v.value
ipv4_send_bytes.clear()
```

as for loop is running, kernel continues with updates, clear() throws those out.

Flow monitoring: system architecture



Evaluation: CPU overhead

using *perf* and FlameGraph[1]

- To record: `perf record -a -g -e cycles -c 5000000 -- sleep 60`
- Post-process: `perf script | FlameGraph/stackcollapse-perf.pl > raw.txt`
- Analyze: `grep -E '(cleanup_module|flowmill_agent)' raw.txt | FlameGraph/flamegraph.pl > flame.svg`

→ observed **0.1% - 0.25% CPU overhead** across deployments

Most aggressive customer load test:

	Node	Application	TCP stack	Collector
M cycles (%)	480,000 (100%)	220,775 (46%)	27135 (5.6%)	4,120 (0.86%)

[1] github.com/brendangregg/FlameGraph

Evaluation: Network overhead

Flow observability → monitor the flow-telemetry flows

Megabytes / second

	App throughput	Flow telemetry	%
Cluster 1	186.2	0.85	0.46%
Cluster 2	217.1	2.49	1.15%
Cluster 3	249.6	0.25	0.10%
Cluster 4 (batch)	522.0	0.16	0.031%
Cluster 5	183.0	0.02	0.013%

→ Usually < 0.5% network overhead, outliers ~1%

Evaluation: Backend QPS

Agent event counts (per second):

	TCP	UDP	NAT	process	container	DNS	Total events/s per agent
Company A	1429.2	82.0	20.8	146.5	0.014	10.5	1689.014
Company B	4017.3	89.0	-	1562.1	-	1.98	5670.38
Company C (batch)	51.0	28.8	1.05	43.8	0.55	0.5	125.7

→ For a 50-node cluster, need to process 84.4k-283.5k QPS
(~20x less for batch workloads)

→ C++ analysis pipeline: hundreds of nodes w/2 second latency
(thousands soon)

Evaluation

- CPU:
  - observed **0.1% - 0.25% CPU overhead** across deployments
  - 0.86% max load test
- Network:
  - Usually < 0.5% network overhead, outliers ~1%
- QPS:
  - ~100k QPS for 50-node cluster
  - can handle 100s of nodes with 2 second latency

Getting application error codes

- eBPF supports user probes

```
$ go tool nm /root/hello | grep 'net/http\.'
  690a40 t net/http.Error
  64eee0 t net/http.Get
  6929e0 t net/http.HandleFunc
  6b6230 t net/http.Handler.ServeHTTP-fm
  6909e0 t net/http.HandlerFunc.ServeHTTP
  6805b0 t net/http.Header.Add
  680700 t net/http.Header.Del
  680690 t net/http.Header.Get
  680620 t net/http.Header.Set
  680750 t net/http.Header.Write
  681190 t net/http.Header.WriteSubset
  680840 t net/http.Header.clone
```

```
$ /funccount -p 31328 '/root/hello:net/http.*Header*'
Tracing 111 functions for
"/root/hello:net/http.*Header*"... Hit Ctrl-C to end.
^C
FUNC                                    COUNT
net/http.Header.Del                         3
net/http.Header.sortedKeyValues             3
net/http.Header.WriteSubset                 3
net/http.(*response).WriteHeader            3
net/http.extraHeader.Write                  3
net/http.(*chunkWriter).writeHeader         3
net/http.(*chunkWriter).writeHeader.func1        3
Detaching...
```

Flow monitoring

Visibility into **Architecture, Health, and Cost**

- No code changes
- Negligible overhead
- Visibility into external dependencies
- Want application metrics (in progress)

} with eBPF

Questions? (and please reach out)

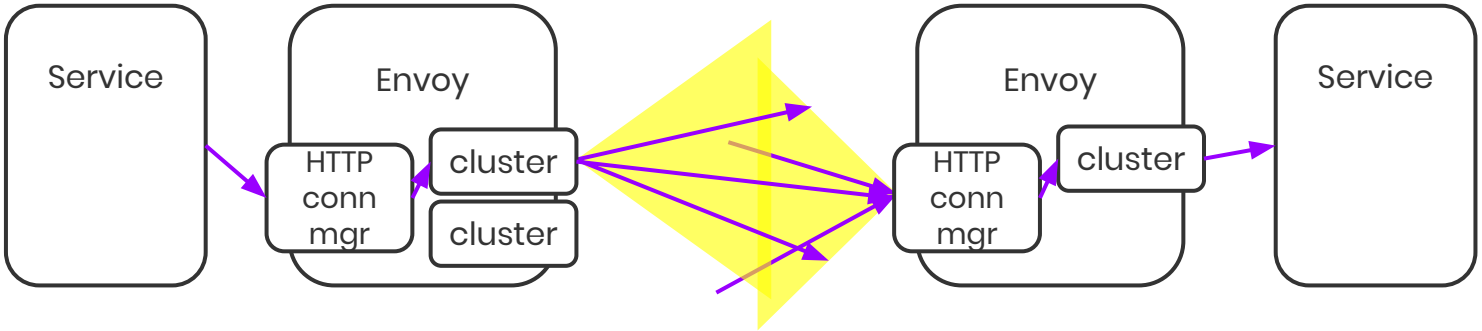
Jonathan Perry <jperry@flowmill.com>
www.flowmill.com

FLOWMILL

Backup slides

What about service mesh?

- Two types of data:
  - Aggregated: missing info on failure domains
  - Access logs: firehose, still need to process 100k+ events/sec



- misconfigured mesh → broken telemetry.
  - want redundant telemetry
- partial deployments & managed services

https://www.youtube.com/watch?v=xZHJQFMVIfY&list=PLWNb_OjvOJz8nTnOVP7a4tSnzWxeJJdpL&index=18&t=0s