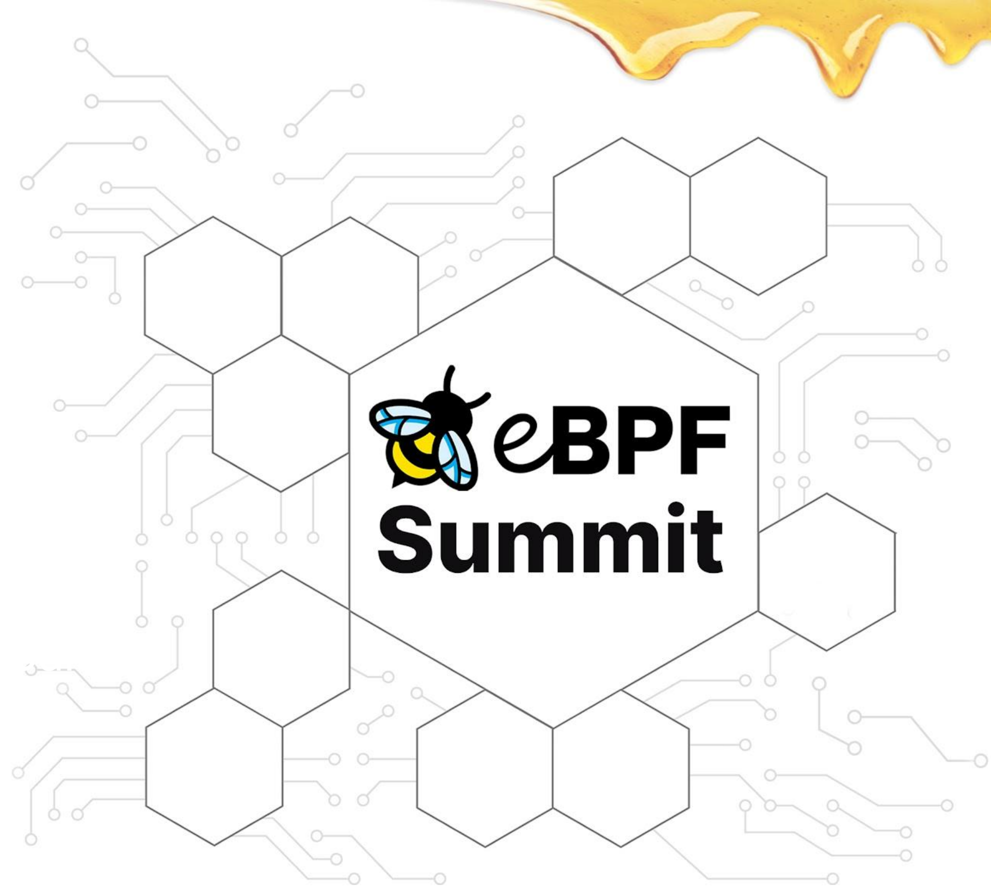
# eBPF: innovations in cloud native




Daniel Borkmann

daniel@cilium.io

What actually is ‘cloud native’?

# Cloud Native: Definition

Cloud native computing is an approach in software development that utilizes cloud computing to "build and run **scalable** applications in **modern**, **dynamic** environments such as **public, private, and hybrid clouds**". Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

[ CNCF Cloud Native Definition v1.0 ]

# Cloud Native: Definition

These techniques enable **loosely coupled** systems that are **resilient**, **manageable**, and **observable**. Combined with robust automation, they allow engineers to **make high-impact changes frequently** and predictably with minimal toil. [...]

[ CNCF Cloud Native Definition v1.0 ]

# Cloud Native: Kubernetes

By 2025, Gartner estimates that over 95% of new digital workloads will be deployed on cloud-native platforms, up from 30% in 2021.

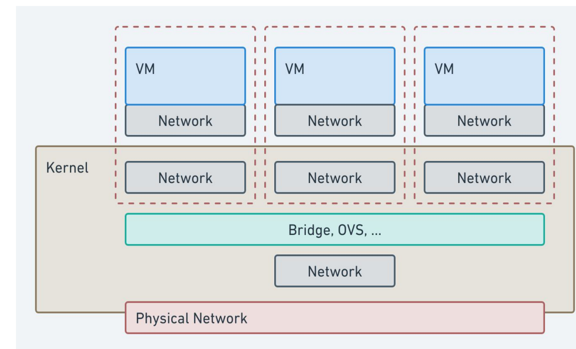
Kubernetes has become the de facto standard for cross-cloud orchestration and [in general] a pillar of cloud architectures.

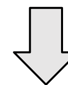
[ Gartner Says Cloud Will Be the Centerpiece of New Digital Experiences ]

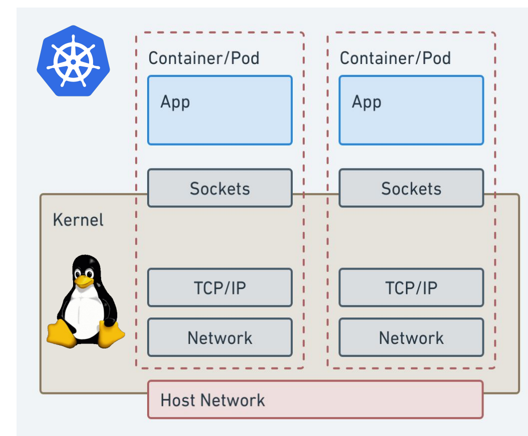
[ Gartner 2022 Planning Guide for Cloud and Edge Computing ]

# Cloud Native: Container Model

- New Networking models: Single kernel becomes common denominator managing many networking objects. Network namespace based, no full-blown VMs, potentially lightweight ones like Kata containers, etc.
- Scale and Scope changing: A few VMs to many containers. Higher per-node container density for efficient resource use. Shorter container lifetime. Dynamic IP pools for containers with high IP churn.
- Generic worker nodes with co-located specialized functions. Entire cluster becomes service load-balancer, policy enforcer, *-mesh.



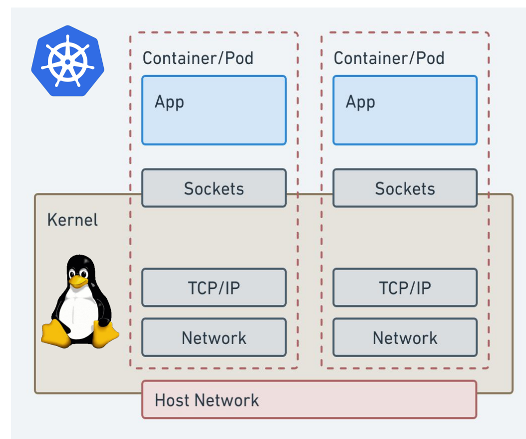



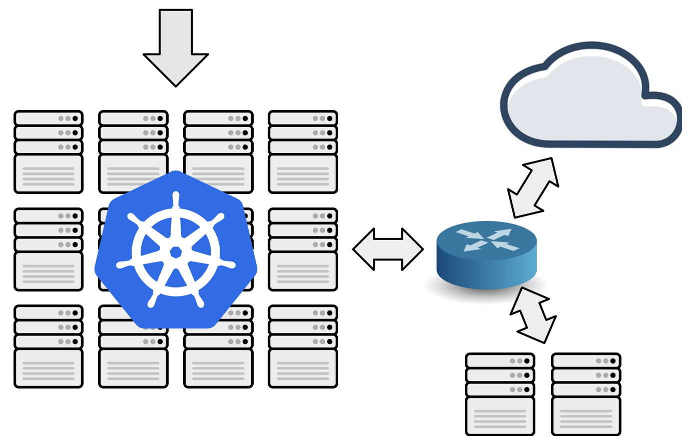


The challenges do not stop there …

# Cloud Native: 'Day 2' Challenges

- Integration requirements with external workloads, e.g. through more predictable IP addresses via service abstractions or egress gateways. BGP for Pod CIDRs, services, gateways.
- Successive migration towards IPv6-only clusters for better IPAM flexibility. NAT46/64 for interaction with legacy workloads.
- Connecting multiple clusters on/off-prem in a scalable manner. Topology aware routing. Traffic encryption.
- Observability and scalable container network and runtime security enforcement.
- Improved network throughput/latency, e.g. for 100G+.



But what about the building blocks?

# Cloud Native: Common Denominator

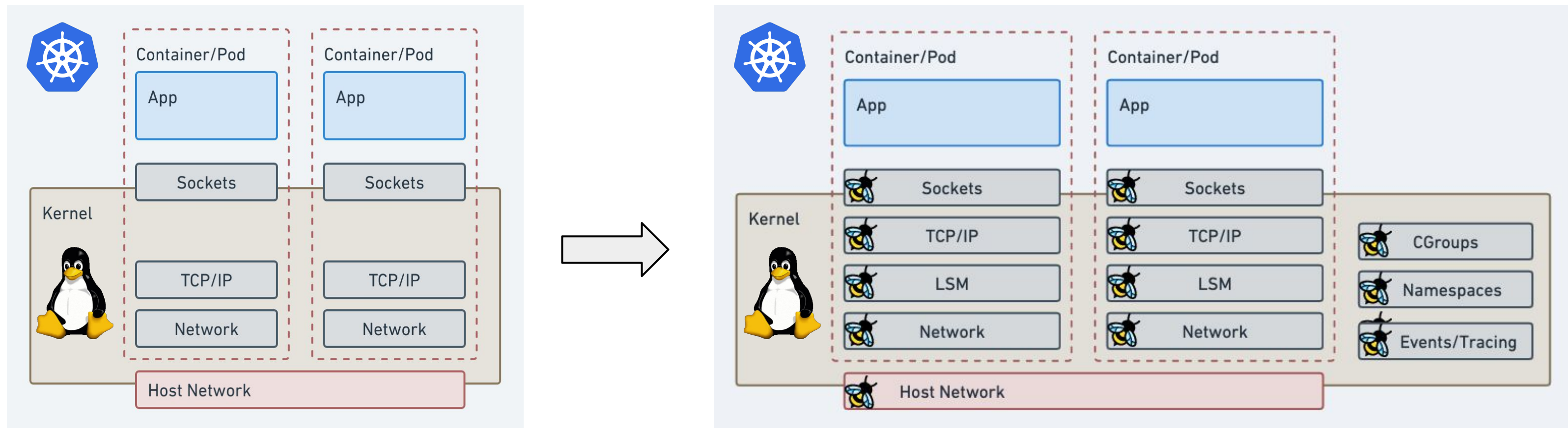
- Linux kernel as foundation with application sockets as data source/sinks and network as communication bus. Linux + Kubernetes as “cloud OS”.
- Building blocks: cgroups (CPU, memory), {net, mount, pid, cgroup, *} namespaces, capabilities, SELinux, seccomp, netfilter, netlink, AppArmor, auditd, perf, etc.
- Often loosely coupled, inflexible (i.e. node-global), inefficient given designed decades ago (i.e. O(n)-style processing) and no Pod-awareness.
- Developers stuck in this ‘box’ given intent cannot be expressed with traditional kernel framework. Not scalable and on top often hard to troubleshoot for users/devops.

eBPF as cloud native tool to innovate

# Cloud Native & eBPF

**Framework to extend the OS kernel**

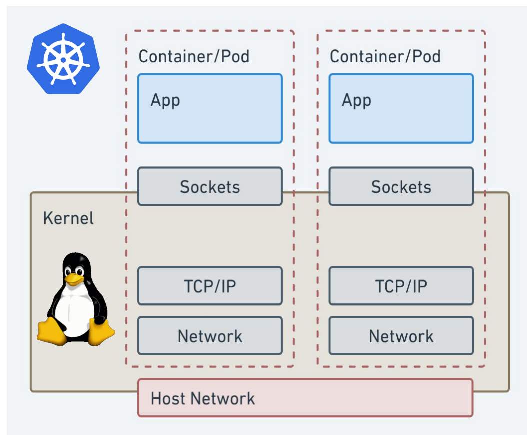
➔ BPF as a general purpose engine with minimal instruction set
➔ Allows for running programs in kernel to customize its behavior without changing kernel's source
➔ Programs atomically updateable to avoid workload disruption and node reboot
➔ Programs verified for safety at load time to prevent kernel crashing or other instabilities
➔ Programs can be made portable between different kernel versions

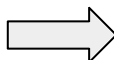


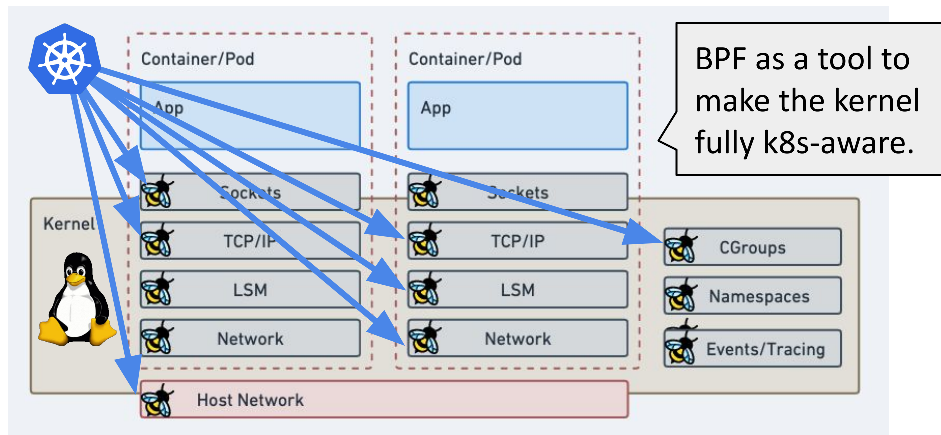
# Cloud Native & eBPF

**Framework to extend the OS kernel**

- ➔ BPF as a general purpose engine with minimal instruction set
- ➔ Allows for running programs in kernel to customize its behavior without changing kernel's source
- ➔ Programs atomically updateable to avoid workload disruption and node reboot
- ➔ Programs verified for safety at load time to prevent kernel crashing or other instabilities
- ➔ Programs can be made portable between different kernel versions





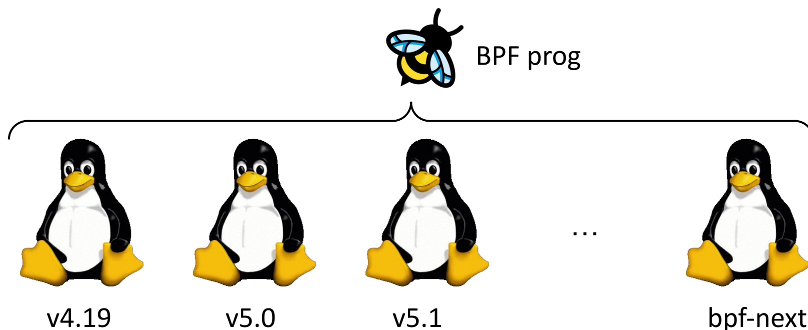

BPF as a tool to make the kernel fully k8s-aware.

# Cloud Native & eBPF: Innovations

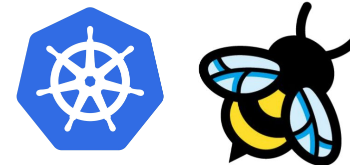
**#1 BPF allows to significantly speed up the development**

➔ Patch lifecycle from development, merge upstream to the point where major distributions get released with it *and* users adopt it can take long time. Production users typically stick only to LTS kernels.

- Example: Ubuntu releases v4.15 (18.04 LTS), v5.4 (20.04 LTS), v5.10 (22.04 LTS, latest) have 2 year cadence. Delta from latest LTS to latest vanilla is 10 kernel releases.
- **Innovation** in the traditional model requires kernel modules or building own kernels, leaving most of the community out. Minimal or no feedback loop from developers to users.
- BPF managed to **break this long cycle** by decoupling from kernel releases, for example, changes in Cilium can be upgraded on the fly on the running kernel and work on large range of kernel releases.

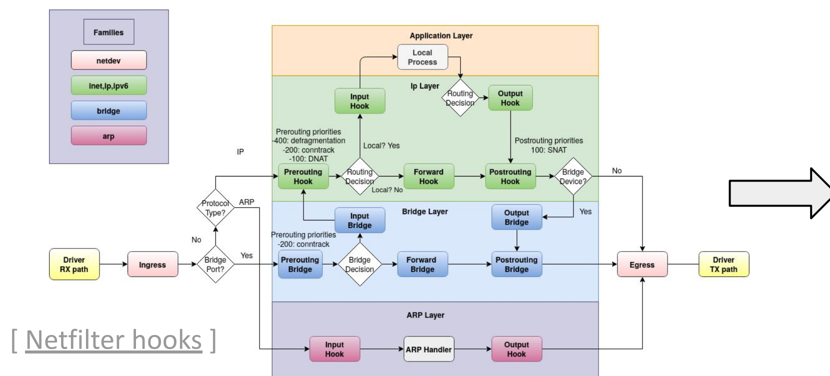


# Cloud Native & eBPF: Innovations

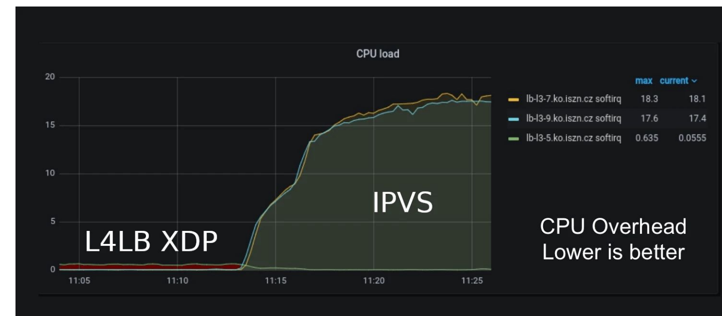


**#2 BPF allows to shift data processing closer to the source freeing up resources**

- ➔ Traditional virtualized networking functions such as LBs/firewalls/etc are solved at a packet level. *Every* packet needs to be inspected, modified or dropped. → $$$
- ➔ **Reframing original problem** by moving further up or down the stack - as close to the event source as possible - with more OS context.
- ➔ Example: BPF shifts towards per-socket hooks, per-cgroup hooks, XDP
  - ◆ Significant **resource cost savings** which allows to migrate from dedicated boxes to generic worker nodes with co-located functions.




[ Netfilter hooks ]




[ seznam.cz on Cilium’s XDP L4LB ]

# Cloud Native & eBPF: Innovations

**#3 BPF allows for shorter production feedback loops and location aware processing**

- ➔ Traditional approach required: patching in-house kernel, gradually rolling kernel to the fleet to deploy change, only then to **start experiment**, collecting data and starting over.
  - ◆ Very long and fragile cycle e.g. nodes need to restart and drain their traffic. Unable to "move fast".
- ➔ Same principle: **decoupling from kernel** and allowing for atomic program updates on the fly
  - ◆ Short feedback loop especially relevant for improvements in datacenter networking
  - ◆ Example: TCP congestion control tuning in BPF (DCTCP, BBR), destination-specific cong. control in BPF
  - ◆ Similarly this is required for process schedulers in BPF

```
__u32 BPF_STRUCT_OPS(bictcp_recalc_ssthresh, struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        ca->epoch_start = 0;    /* end of epoch */

        /* Wmax and fast convergence */
        if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
                ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
                        / (2 * BICTCP_BETA_SCALE);
        else
                ca->last_max_cwnd = tp->snd_cwnd;

        return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
}
```

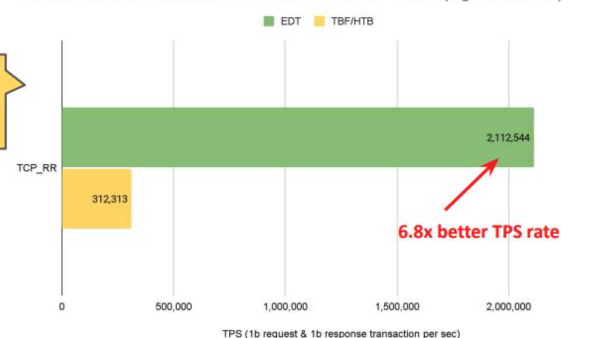
```
static u32 bictcp_recalc_ssthresh(struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        ca->epoch_start = 0;    /* end of epoch */

        /* Wmax and fast convergence */
        if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
                ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
                        / (2 * BICTCP_BETA_SCALE);
        else
                ca->last_max_cwnd = tp->snd_cwnd;

        return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
}
```

[ BPF Extensible Network: BPF congestion control (left) vs. native kernel congestion control (right) ]

# Cloud Native & eBPF: Innovations

**#4 BPF enables to move traffic with significantly lower latency**

- ➔ Allows to implement node-local forwarding infrastructure for **datacenter workloads** via tc BPF layer
  - Example: Cilium contains BPF host routing and BPF bandwidth manager
  - Retains packet's socket association and egress timestamps *all the way* from Pod to phys device
  - Main theme where BPF helps is around achieving host-native performance for Pods (netns'es)
- ➔ Foundation for TCP pacing, TCP BBR congestion control, TCP TSQ, and EDT rate-limiting for Pods
  - All tie into **reducing Pod's bufferbloat** in the host stack and incast in the datacenter network




[ Better Bandwidth Management with eBPF: HTB (yellow) vs. BPF + EDT (green) ]

# Cloud Native & eBPF: Innovations

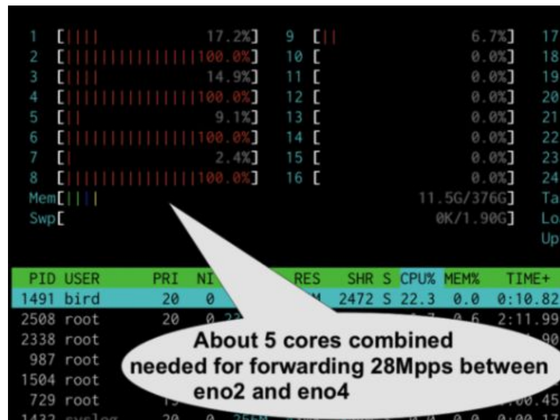
**#5.1 BPF provides building blocks from kernel instead of reinventing the wheel**

- ➔ Example: BPF fib lookup helper reusing kernel routing and neighbor tables
  - ◆ User space BGP stacks like FRR can all **transparently be reused** and routers built out of XDP layer
  - ◆ Allows for forwarding plane solely in tc BPF layer via bpf_redirect_{peer,neigh} helpers
- ➔ In fact, most components and system tooling can be reused as-is
  - ◆ Even AF_XDP just piggybacks on upstream drivers, netdevs, NAPI
  - ◆ Enables easy integration **without rewriting larger parts** of the user space stack

```
BGP state:            Established
  Neighbor address:   147.75.194.163
  Neighbor AS:        1
  Neighbor ID:        147.75.194.163
  Neighbor caps:      AS4
  Session:            internal multihop AS4
  Source address:     10.99.204.3
  Hold timer:         117/180
  Keepalive timer:    5/60

root@xdp:~# ip route | wc -l
807195
```

Linux routing table / FIB has 800k routes

[ Build an XDP based BGP peering router ]



# Cloud Native & eBPF: Innovations

**#5.2 BPF provides building blocks which are too complex for other kernel subsystems**

- ➔ Example: Netfilter subsystem *never* had a NAT46/64 translation possibility
  - ◆ Complex IPv4/IPv6 code-base and kernel's packet representation (skb) contains address family specific data, e.g. around GRO/GSO. Some out-of-tree PoCs Jool, CLAT but they never made it upstream.
- ➔ For Cilium, we recently added **NAT46/64 via BPF** to connect IPv6-only K8s cluster via NAT46/64 gateway
  - ◆ XDP BPF side is fairly straight-forward and most efficient given skb does not exist there
  - ◆ tc BPF side done through bpf_skb_change_proto helper to alter skb internals for tc layer
    - • Also used by Android via BPF today to connect phone to IPv6-only cell network

```
cilium service list
ID   Frontend      Service Type   Backend
1    1.2.3.4:80    ExternalIPs    1 => [f00d::1]:60 (active)
                                  2 => [f00d::2]:70 (active)
                                  3 => [f00d::3]:80 (active)
```

```
cilium service list
ID   Frontend        Service Type   Backend
1    [cafe::1]:80    ExternalIPs    1 => 1.2.3.4:8080 (active)
                                    2 => 4.5.6.7:8090 (active)
```

[ Cilium 1.12: NAT46/64 for L4LB ]

# Cloud Native & eBPF: Innovations

**#6 BPF (to some degree) allows to fix or mitigate kernel bugs on the fly**

- ➔ Recently used to fix a kernel bug in veth driver affecting ena's TX queue selection
  - See also Laurent's summit talk: All Your Queues Are Belong to Us
  - On-the-fly fix **avoided complex rollout of new kernel**, this time from cloud provider side which can take even longer given out of hands from infrastructure team.
- ➔ **Resilient receive processing** to reduce attack surface from bad actors
  - Complex in-kernel flow dissector can be replaced entirely in BPF or augmented partially in BPF
  - XDP as packet-of-death mitigation given stack (GRO and above) did not see packet yet

1 Open ✓ 11 Closed | Author ▾ | Label ▾ | Projects ▾ | Milestones ▾ | Reviews ▾ | Assignee ▾ | Sort ▾

bpf: Reset Pod's queue mapping in host veth to fix phys dev mq selection ✕ backport-done/1.9 backport-done/1.10 backport-done/1.11 release-note/misc — 8
#18388 by borkmann was merged on 6 Jan • Approved

[ https://github.com/cilium/cilium/pull/18388 ]

# Cloud Native & eBPF: Innovations

**#7 BPF enables low-overhead deep visibility and enforcement into the system**

- ➔ For building low-overhead fleet-wide tracing/observability platforms with BPF collector(s) on each node
- ➔ For troubleshooting production issues on-the-fly in a safe way, e.g. via bpftrace
  - ◆ Both allow for significantly richer visibility, programmability and ease-of-use than old-style perf

## Example: bpftrace biolatency

Implemented in <20 lines of bpftrace

```
#!/usr/local/bin/bpftrace

BEGIN
{
        printf("Tracing block device I/O... Hit Ctrl-C to end.\n");
}

kprobe:blk_account_io_start
{
        @start[arg0] = nsecs;
}

kprobe:blk_account_io_done
/@start[arg0]/
{
        @usecs = hist((nsecs - @start[arg0]) / 1000);
        delete(@start[arg0]);
}
```

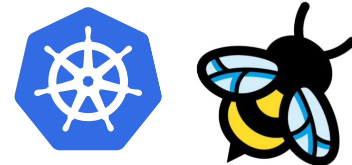
[ BPF Performance Analysis at Netflix ]

To collect the histogram of CWND for currently sent TSO frames, normal queues vs the overloaded queue (queue 3 in the example) we used:

```
tracepoint:net:net_dev_start_xmit {
if (args->gso_type != 16) {return;} /* skip non-TSO */

$skb = (struct sk_buff *)args->skbaddr;
$tp = (struct tcp_sock *)$skb->sk;
$icsk = (struct inet_connection_sock *)$tp;

@[$icsk->icsk_ca_ops,
  $skb->sk->__sk_common.skc_rx_queue_mapping == 3] =
    hist($tp->snd_cwnd);
}
```

[ Investigating TCP Self-Throttling Triggered Overload: bpftrace example ]

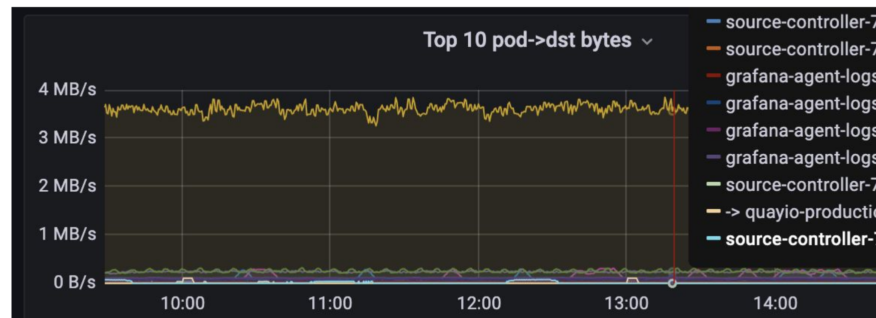
# Cloud Native & eBPF: Innovations

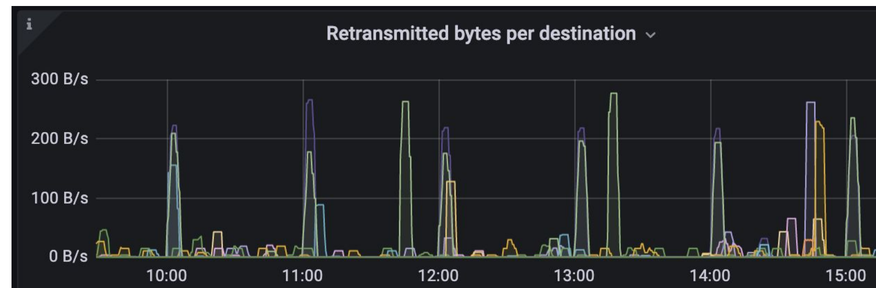


**#7 BPF enables low-overhead deep visibility and enforcement into the system**

```yaml
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "kill_unprivileged_user_namespace"

# Restricts access to Linux user namespace functionality. Any unprivileged
# (without CAP_SYS_ADMIN) that tries to create a user namespace will be
# killed.

spec:
  kprobes:
  - call: "create_user_ns"
    syscall: false
    args:
    - index: 0
      type: "nop"
    selectors:
      - matchCapabilities:
          - type: Effective
            operator: NotIn
            isNamespaceCapability: false
            values:
              - "CAP_SYS_ADMIN"
        matchActions:
        - action: Sigkill
          argError: -1
```

[ https://github.com/cilium/tetragon/pull/399 ]



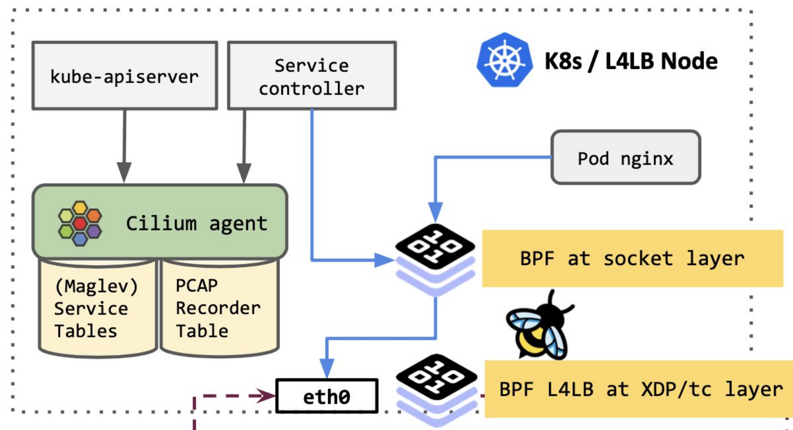




[ Tetragon: Grafana Dashboard examples ]

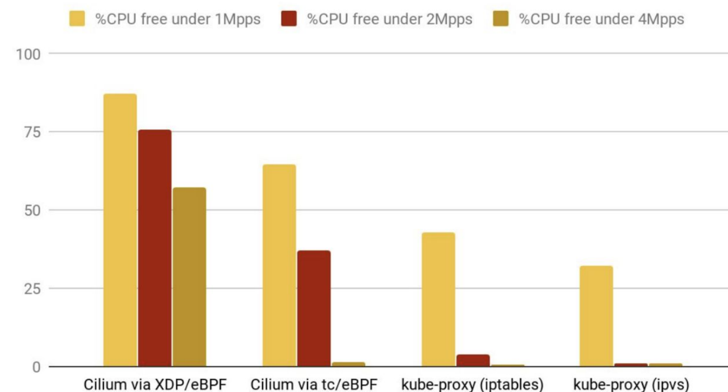
# Cloud Native & eBPF: Innovations

**#8 BPF decouples from legacy UAPI and allows for efficient data processing**

- ➔ Reduces the kernel's feature creeping normality and keeps fast-path to a minimum and fast
- ➔ Complex, custom use-cases don't need to become kernel UAPI, just the building blocks in BPF
  - ◆ Building blocks like helpers, maps are also decoupled from entry-point into BPF
  - ◆ Example: Fully conformant, faster and customizable kube-proxy replacement in BPF




[ Kubernetes service load-balancing at scale with BPF & XDP ]




[ kube-proxy via iptables/ipvs vs BPF: CPU capacity ]

# Cloud Native & eBPF: Innovations

**#9 BPF is an enabler to build policy enforcement features around stronger notions of identity**

➔ Abstracting away from **high Pod IP churn** is crucial in Kubernetes environments
- IPs become meaningless given everything is centered around Pod labels and generally Pod lifetime is decreasing, e.g. ephemeral workloads
- BPF helps to abstract from IP to identity and also to build egress gateways for Pods with more stable IPs

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l3-egress-rule"
spec:
  endpointSelector:
    matchLabels:
      role: frontend
  egress:
  - toEndpoints:
    - matchLabels:
        role: backend
```

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l4-rule"
spec:
  endpointSelector:
    matchLabels:
      app: myService
  egress:
    - toPorts:
      - ports:
        - port: "80"
          protocol: TCP
```

```yaml
apiVersion: cilium.io/v2
kind: CiliumEgressGatewayPolicy
metadata:
  name: egress-sample
spec:
  selectors:
    - podSelector:
        matchLabels:
          app: test-app
  destinationCIDRs:
    - 192.0.2.0/24
  egressGateway:
    nodeSelector:
      matchLabels:
        kubernetes.io/hostname: 'myEgressNode'
    # specify either 'egressIP' or 'interface':
    egressIP: '192.0.2.2'
```

[ Cilium: Identity-based Security ]

# Cloud Native & eBPF: Innovations

**#10 BPF allows developers to extend kernel but with a safety-belt on**

- ➔ Perhaps the most important feature that BPF brings to the table
  - ◆ Development and testing costs much higher for kernel code vs BPF code for *same* functionality
  - ◆ Also troubleshooting production environments easier due to BPF's built in safety via verifier
  - ◆ Portability for BPF modules across kernel versions is achieved with CO-RE, kconfigs, BTF type info
- ➔ BPF flavor of the C language is a better and safer choice for kernel programming

```
int err_cast(struct task_struct *tsk)
{
   return((struct sk_buff *)tsk)->len;
}

int err_release_twice(struct __sk_buff *skb)
{
    struct bpf_sock_tuple tuple = {};

    struct bpf_sock *sk = bpf_sk_lookup_tcp(skb, &tuple, sizeof(tuple), 0, 0);
    bpf_sk_release(sk);
    bpf_sk_release(sk);
    return 0;
}
```

OK in C.
NOT OK in BPF C.

[ The journey of BPF from restricted C language towards extended and safe C. ]

What comes next?

# Cloud Native & eBPF: Future

**#1 BPF core: More building blocks and lower entry barrier for developers**

- ➔ On the way to support 100% of C in safe way... today BPF already supports:
  - Global and static variables
  - Global and static functions
  - Type information via BTF for symbolic access (CO-RE) and safety analysis
  - Loops (bounded loops, bpf_loop helper, bpf_for_each* helpers, iterators)
  - Atomics, timers, spinlocks
- ➔ Extensions to BPF core is production need driven
  - New use cases such as BPF process scheduler also push boundaries for BPF core, e.g. being able to implement data structures natively within BPF or having programs in place at early boot
- ➔ Cross platform support & standardization
  - Feature parity between all major architectures (x86-64/arm64/riscv64/ppc64/etc), compilers (LLVM/gcc), and platforms (Linux/Windows/hardware/user space)
  - eBPF foundation to publish iterations of BPF specification, guidelines for developers and distributions, and technical roadmap

# Cloud Native & eBPF: Future

**#2 BPF core: Signing and supply chain security**

- ➔ Main goal is the attestation of origin to verify if source is authorized to run BPF programs
  - Keeping out bad actors while also clearly identifying allowed actors using BPF
  - Challenge: the vast majority of programs in the ecosystem are of dynamic nature (think of bpftrace)
  - If signing would only cover static programs, we'd lock out 99% of users which is a clear non-starter
- ➔ Various approaches are under discussion, e.g. container-level or BPF program-level along with all assets to be able to perform CO-RE in-kernel after signature check
  - Not really one-size-fits-all type solution for covering different scenarios
  - Gatekeeper BPF program at early boot with hooks into verifier and BPF syscall commands which performs signature checks via in-kernel key management facilities
  - Signatory service for BPF ecosystem to solve dynamicity
  - Might later also enable granular restrictions
- ➔ Ideally multi-platform support given high interest for supporting Linux/Windows runtime
  - eBPF foundation could provide a reference implementation

# Cloud Native & eBPF: Future

**#3 BPF core: Increased focus on profiling BPF with BPF**

➔ Increased use of various BPF components doesn’t automatically translate to ‘faster’
- Badly written programs can of course cause performance regressions
- More visibility into profiling BPF itself becomes increasingly important
- Early work/ideas around things like `bpftool prog profile` to extract performance counters and code-coverage for programs

# Cloud Native & eBPF: Future

**#4 BPF applications: Cloud native will see even bigger adoption**

- ➔ Similarly as Kubernetes fans out further into edge environments, so does BPF
  - 5G dataplanes with BPF, IoT security enforcement with BPF exist today
  - APM and security monitoring platforms with BPF at its core
  - XDP & BPF L4LBs/gateways into the cloud, memcaches/accelerators, etc
  - Overheard at LPC conference: XDP and BPF is running in wind turbines already
- ➔ Adoption growing in ISP and telco environments as well
  - XDP and AF_XDP blends in naturally with Kubernetes as opposed to DPDK given it suits generic workloads, shipped upstream everywhere, easy to use and context already in kernel space
  - Traffic engineering via BPF and SRv6
- ➔ See also initial Gartner statement with regards to Kubernetes mainstream adoption
  - BPF-based solutions piggy-back on this as well given they are preferred choice or default already
  - Clear trend we also see from users that vast majority is on 5.4/5.10+ kernels by now thanks to AL2, COS, Flatcar, Ubuntu and other cloud based images iterating faster providing good foundation

# Cloud Native & eBPF: Future

**#5 BPF applications: Strong continuous growth of existing and new areas**

- ➔ New use cases on the horizon
  - BPF process scheduler to customize e.g. CFS for data center workloads. Different proposals from Google, Meta, Huawei and convergence around base infrastructure that fits everyone.
  - BPF and IMA for file integrity protection/monitoring of system software and user applications
  - BPF and XDP-like layer plus better observability for storage devices, e.g. around block layer and below
- ➔ Longer-term or moonshot-type directions
  - More use-cases around BPF as safe and portable kernel modules
    - BPF drivers for HID devices set precedence but think more generically
    - BPF could potentially solve the RHEL-like kABI challenge via CO-RE
  - Kernel live patching through BPF which may not be too far out with f{entry,mod_ret,exit}
  - Sustainable computing projects like Kepler exporting power consumption data via BPF for Kubernetes clusters. Also ties into data sharing for richer context between layers, e.g. for BPF process scheduler.

**ebpf.io**