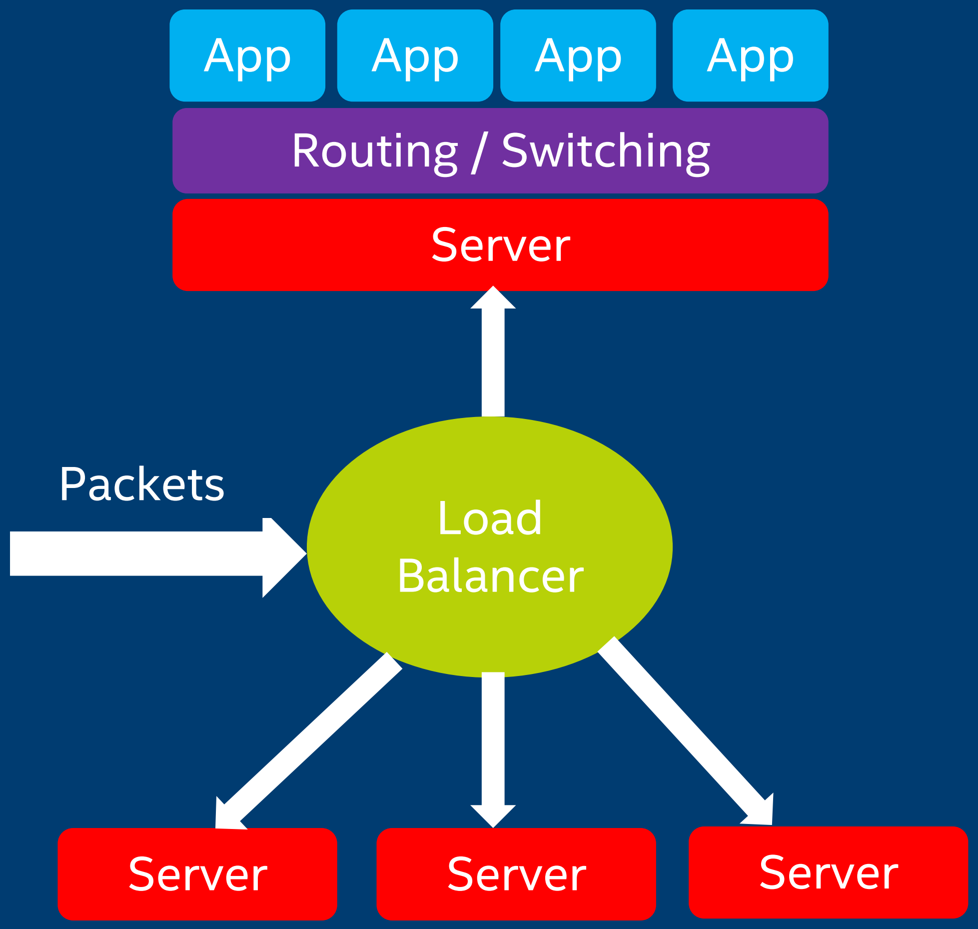
FUNDAMENTAL TECHNOLOGIES TO WORK ON FOR CLOUD-NATIVE NETWORKING

Magnus Karlsson, Intel

Cloud-Native Network Functions – My View

- Many small network functions
- Runs in containers / processes
- High availability
- Automatic scalability
- Secure
- Deployable at scale
    - Really simple
- Load-balancing
- Routing and/or switching
- Best performance NOT a main driver



Cloud-Native systems using the Linux stack is NOT a focus of this presentation

Properties Needed

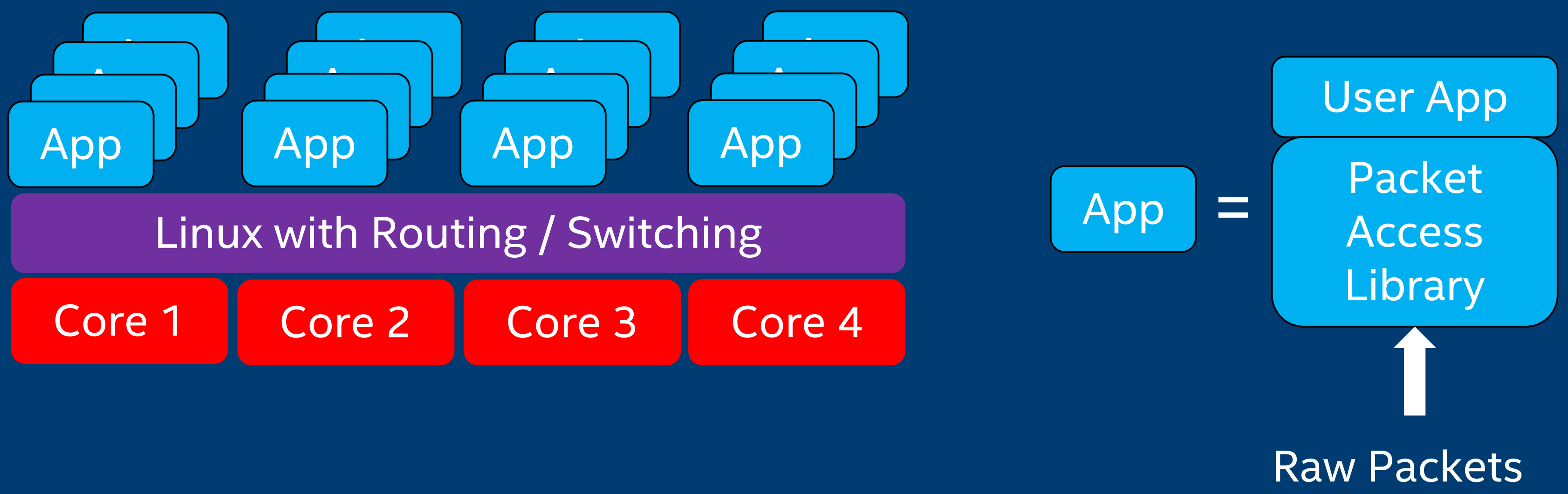
Requirements

- Many small network functions
- Runs in containers / processes
- High availability
- Automatic scalability
- Secure
- Deployable at scale
  - Really simple
- Load-balancing
- Routing and/or switching
- Good enough performance

Properties

- HW agnostic – Linux APIs only
- Fault isolation
- Restartability
- Multiple SW versions
- Upgradeable during run-time
- Many processes per core
- Power save
- All security features working
- Debuggable & observeable
- Routing/switching in kernel
- Binary compatibility
- Works on any standard Linux

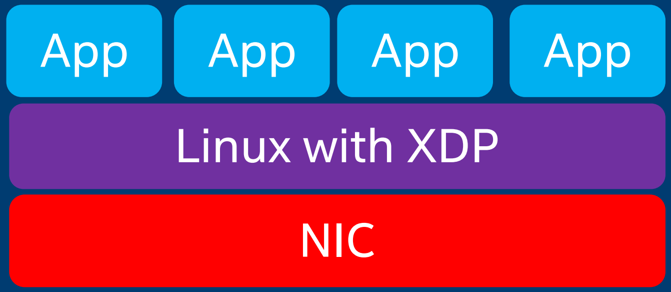
Desired System



<u>All drivers in the Linux kernel the key to solving the problem</u>

Goal for Cloud-Native Dataplane

- Dead-simple, out-of-the-box cloud-native networking for network functions
- With the properties outlines previously
- Supported by all major distributions
- Binary backward and forward compatibilty
- With good enough performance



Features We Cannot Use

DESIRED

- HW agnostic – Linux APIs only
- Fault isolation
- Restartability
- Multiple SW versions
- Upgradeable during run-time
- Many processes per core
- Power save
- All security features working
- Debuggable & observeable
- Routing/Switching in kernel
- Binary compatibility
- Works on any standard Linux

NOT AN OPTION

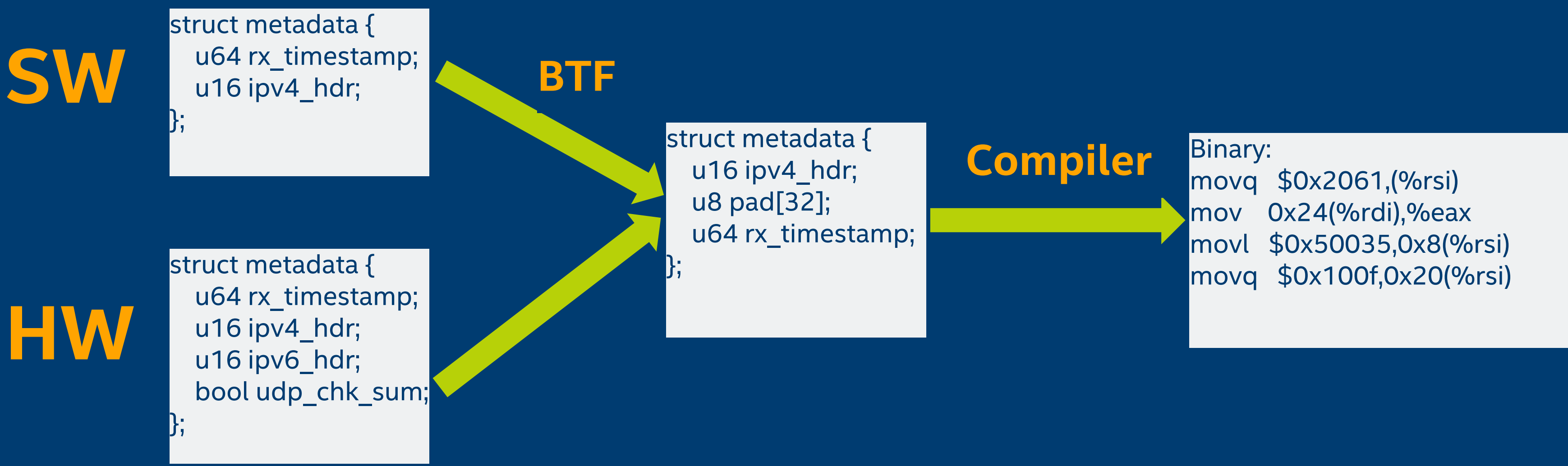
- SR-IOV
- User-space drivers
- Pinned cores & memory
- Busy-polling
- Huge pages
- Shared memory
- 1-to-1 virtual to physical mappings
- >1 crossing user/kernel-space
- Monolithic SW
- Custom kernel modules
- Complete kernel bypass
- Hard-coded platform

Linux NIC features << Features of HW NIC

In Linux we need to develop:

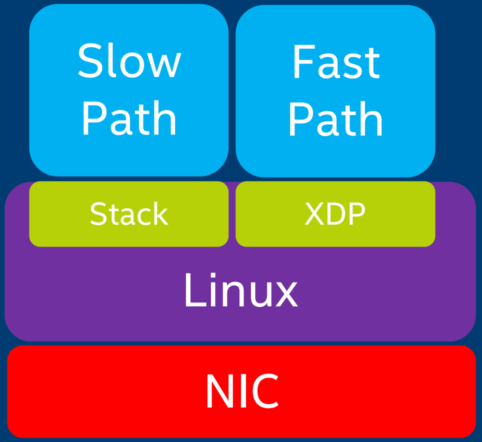
- Metadata and offloadning support for XDP & AF_XDP
    - Supporting accelerators
- Making it easy to orchestrate and control
    - Managing both the fast path and the slow path (Linux networking stack) using the Linux stack control plane
    - Slicing up a netdev with real HW queues
    - Preallocating AF_XDP memory for the containers using Kubernetes
- Queue management
    - For deployment at scale
- Packet access library designed for cloud-native and Linux

Metadata and Offloading



- No mbuf or skbuf needed. Access metadata directly
- Only pay for the metadata you use
- XDP has a JIT, so can be done in run-time
- AF_XDP needs to dynamically link at bind() time or use an offset table
- Accelerators probably will use io_uring. How to support metadata there?

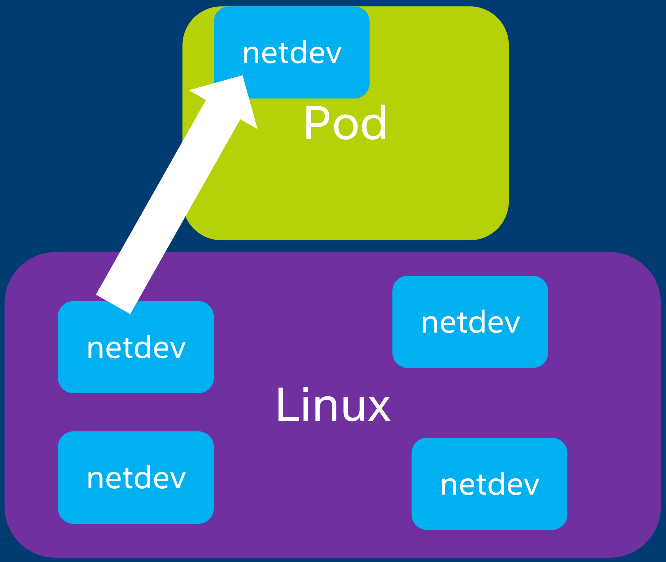
Controlling the Fast Path from Linux




```
xdp_action xdp_program() {
    ip_src = extract_ipv4_src_addr();
    ip_dst = extract_ipv4_dst_addr();
    :
    bpf_route_lookup(ip_src, ip_dst,...);
    route_to_dst();
};
```

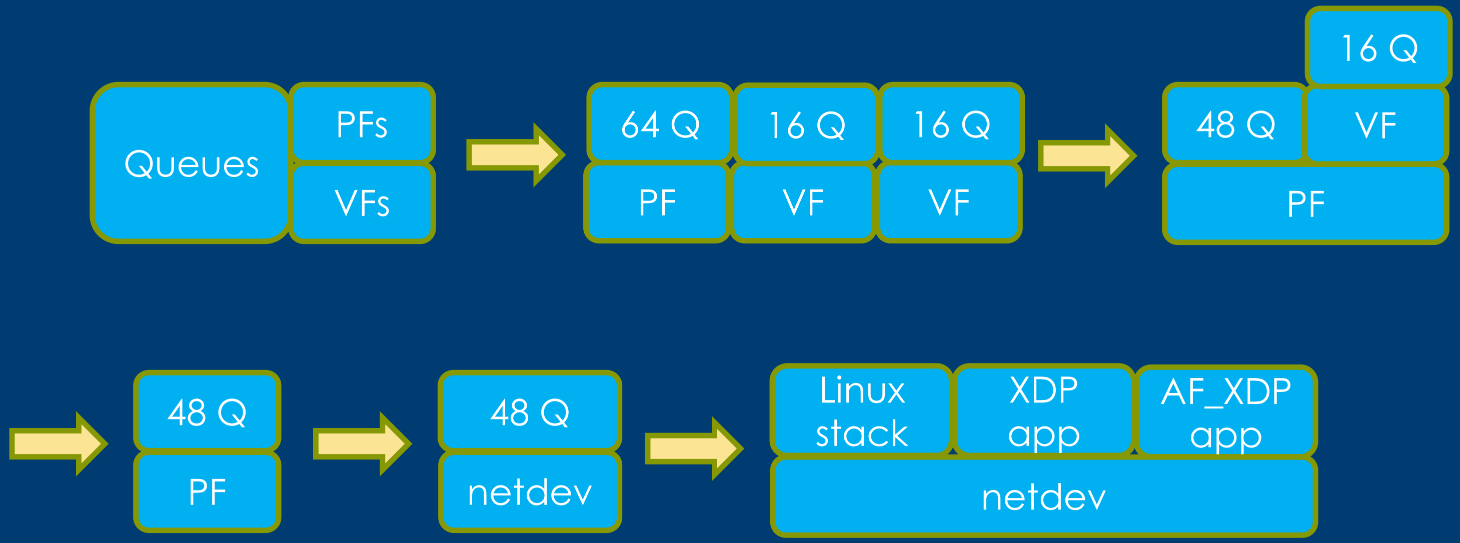
- Linux control path sets up actions in HW and/or XDP
  - XDP when HW does not support the action
- All packets pass XDP
- Use helpers in XDP
  - Reads kernel state or metadata from NIC
  - But not many of these exists today

Facilitating Kubernetes Orchestration



- AF_XDP needs a netdev with real HW queues
  - How to create one of those?
  - Use Macvlan with add_station support?
- Pod needs to have all AF_XDP memory areas preallocated
  - Launch a ”pre-process” that then forks off a child that becomes the pod

Queue Management: The Focus

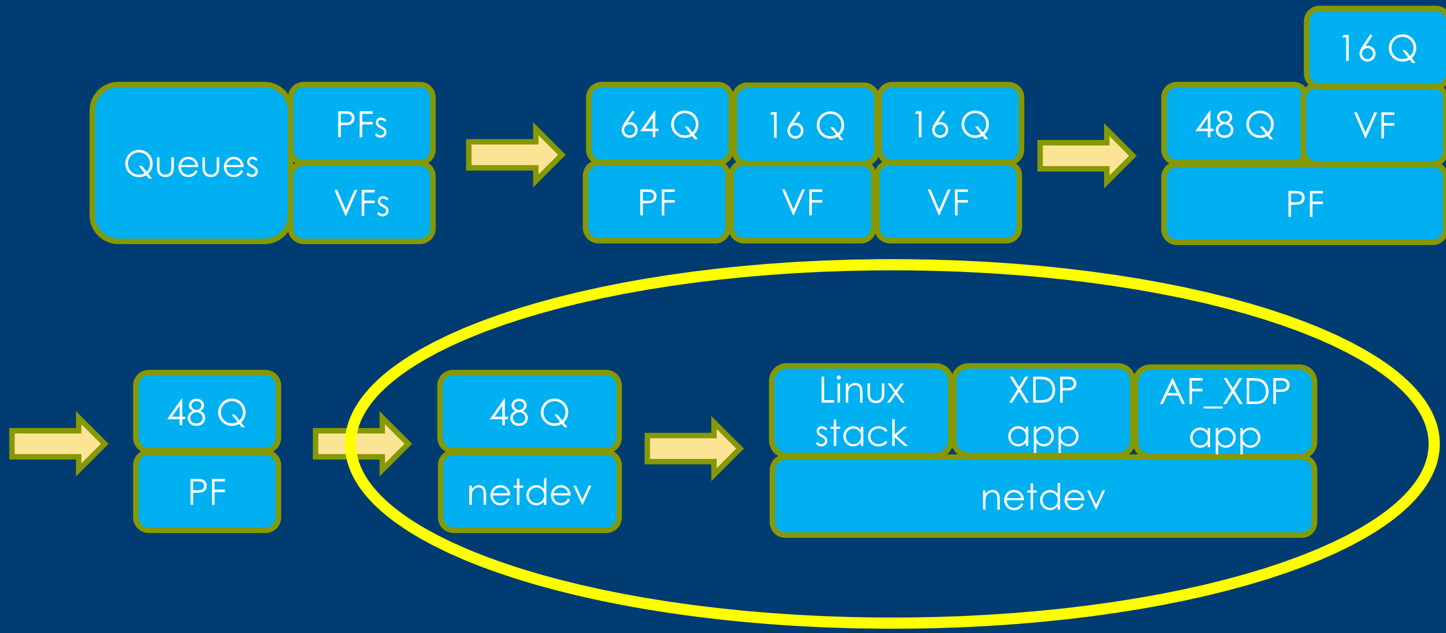


<u>Two problems:</u>

Splitting up queues between PFs and VFs in a device

Allocating and freeing queues within a netdev

Queue Management: The Focus

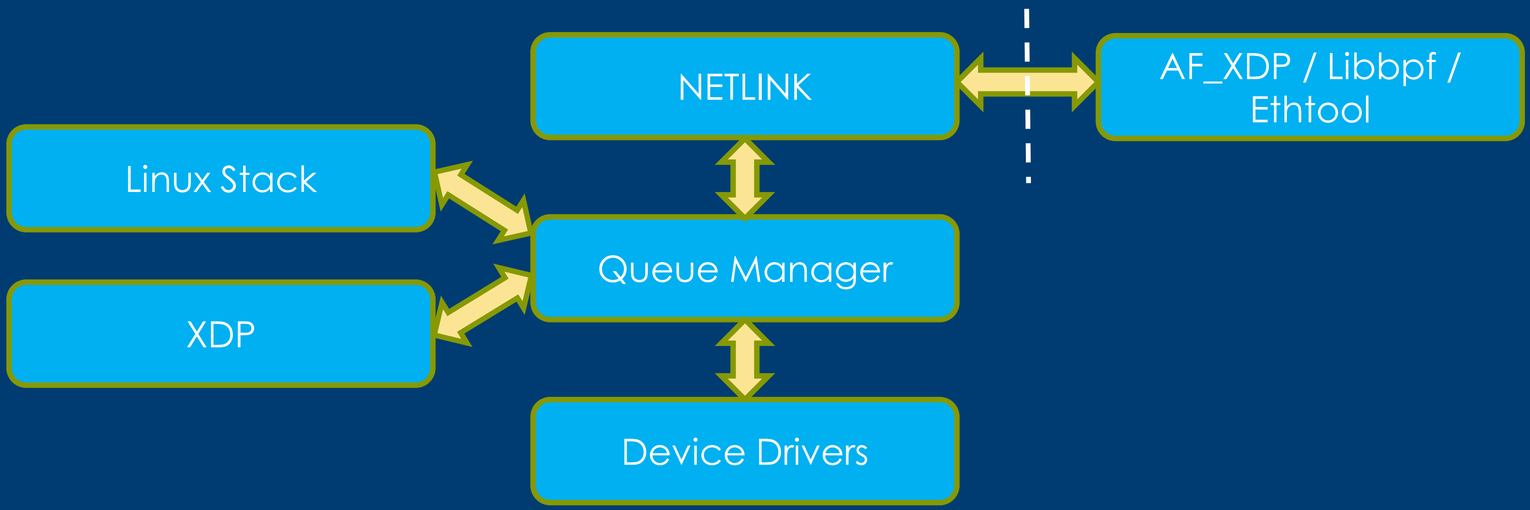


<u>Two problems:</u>

Splitting up queues between PFs and VFs in a device

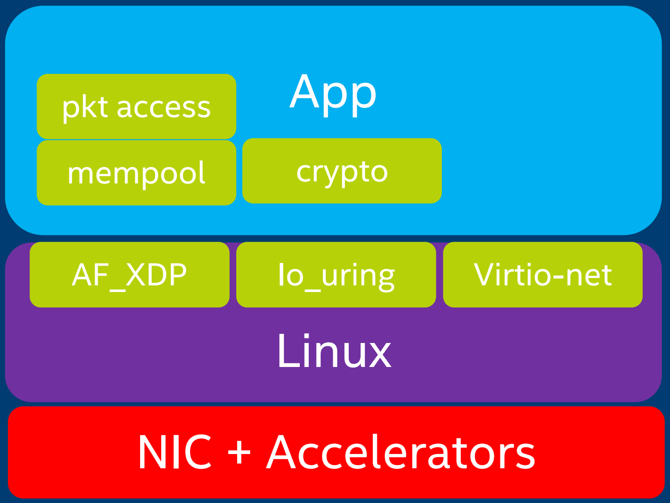
Allocating and freeing queues within a netdev

Kernel Design Overview



- New alloc and free ndo:s in driver needed
  - Tie into existing interfaces, e.g. netif_set_real_num_rx_queues()
- Qids can be decided by driver
  - For backwards compatibility and encoding queue types
- When used in conjunction with netdev slicing => custom netdevs

Cloud-Native Packet Access Library



Important properties:

- All drivers in kernel space
- Set of small shared libraries
- No HW exposed to user space
- Does not force a platform on the users
  - No config, launch, or run-time environment in libraries
  - Works in both processes and threads in any configuration
- No mbuf or the likes exposed to the application
- Applications cannot crash each other
- Debugability, observability and testability from day one
- First optimized for ease-of-use and the right functionality, then optimize for performance

Conclusions

- Cloud-native ≠ appliance or virtual machine
- Most of the challenges solved by having all drivers in the kernel
- But Linux is not ready for this:
  - Metadata and offloading
  - Controlling the data plane from the Linux stack
  - Orchestration support: splitting up netdevs
  - Queue management
- New requirements on packet access libraries
  - Do we evolve DPDK or do we need a new packet library?

experience
what's inside™