MARVELL

DPU-Offloaded TLS Termination and Session Routing for Stateful MCP Traffic

Securing agent-to-tool traffic at scale, deployed with Kubernetes

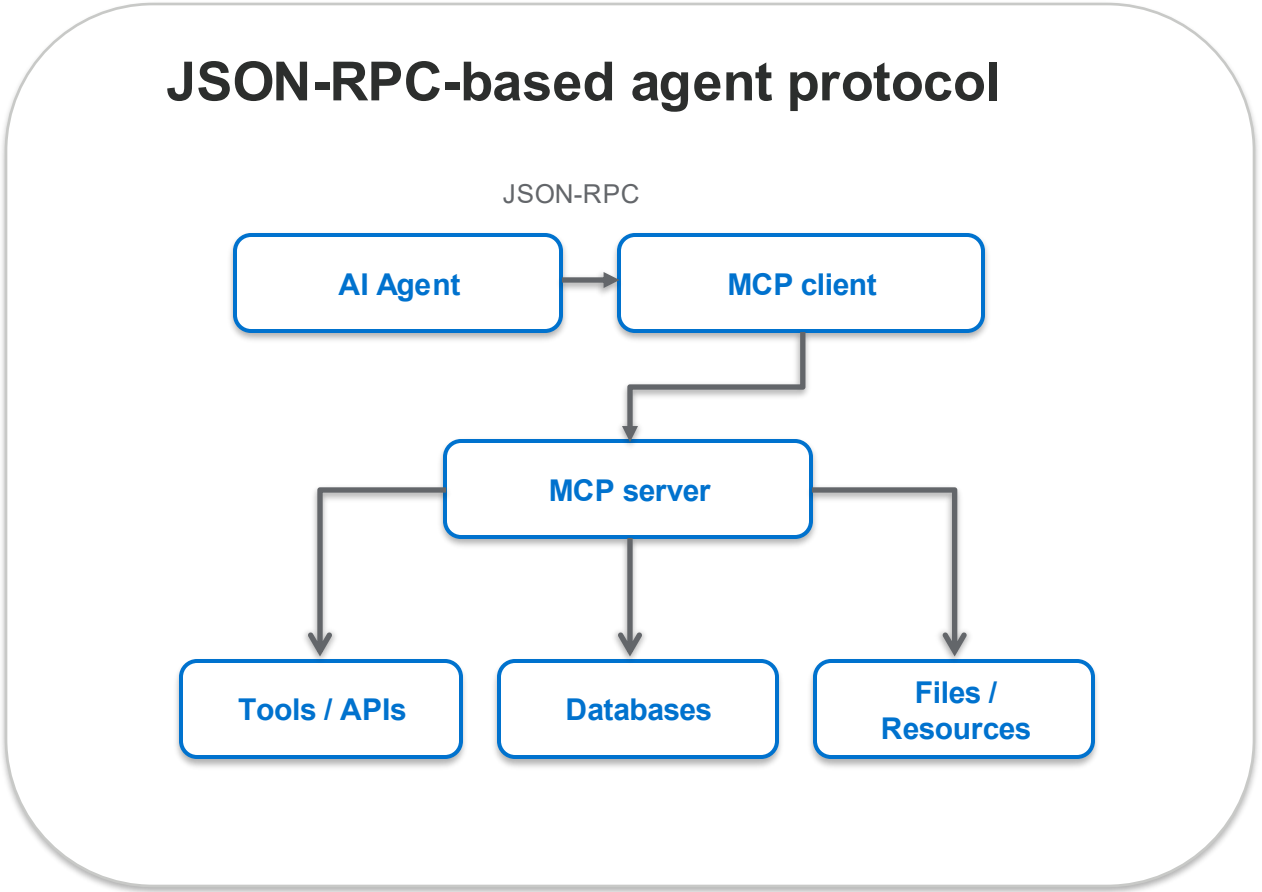
Balakrishna Bhamidipati
bbhamidipati@marvell.com

Vijay Ram Inavolu
vinavolu@marvell.com

NetDevConf 2026

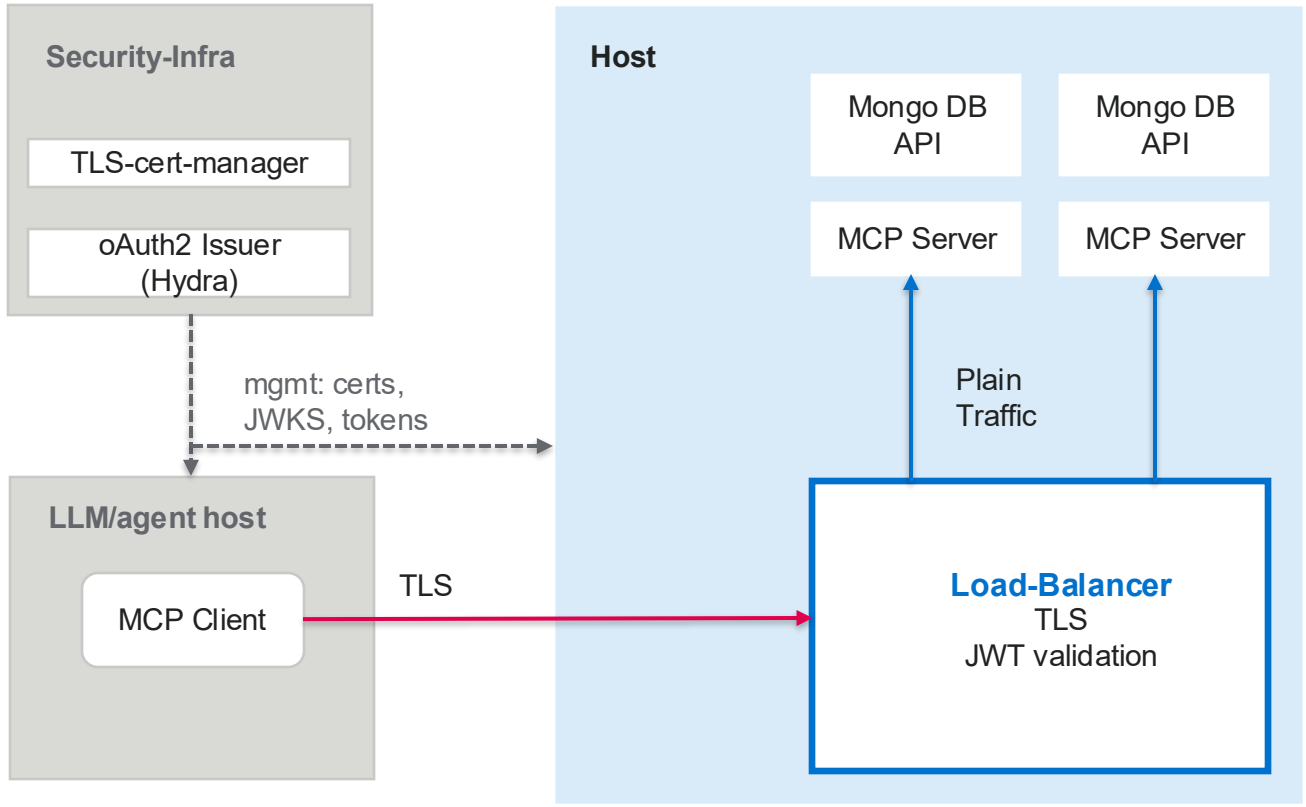
MCP standardizes how AI agents reach external tools

- Model Context Protocol (MCP) defines how AI agents discover external capabilities
- Exposes a consistent surface (`tools/list`, `tools/call`, `resources`, `prompts`) over standard JSON-RPC
- Abstracts backend specifics — one protocol across:
  - APIs & REST services
  - Databases & SQL drivers
  - File systems
  - Enterprise tools
- Without MCP, each new API surface requires bespoke client integration code



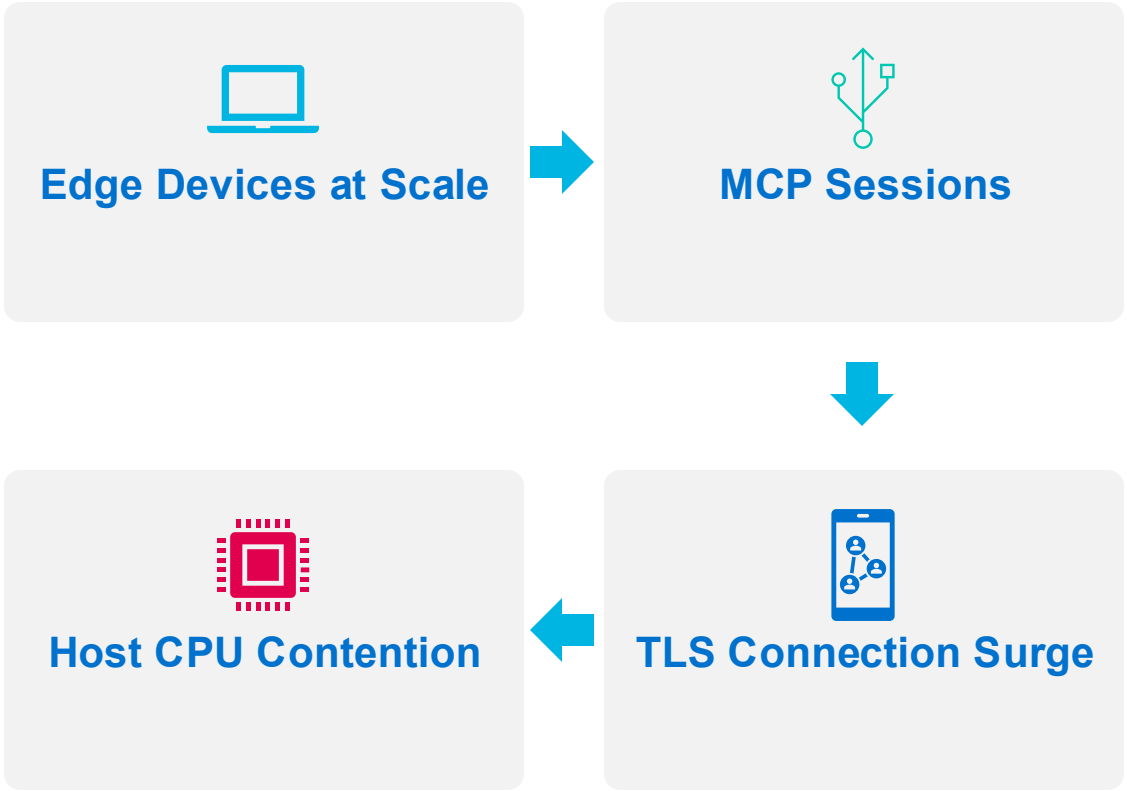
MCP's open surface demands security and authentication

- AI agents access sensitive data, introducing several risks:
  - Data interception
  - Unauthorized clients
  - Malicious MCP servers
- Every agent connection needs a protected path to backends
- Every request must be authenticated and authorized (OAuth 2.0 / JWT)
- Traditionally, security is managed entirely on the host



Scaling MCP turns security into an infrastructure problem

- **Edge devices at scale** — millions of mobiles & laptops run AI apps (ChatGPT, Cursor, Claude)
- **MCP sessions** — each app opens sessions to reach APIs & tools
- **TLS connection surge** — a large number of TLS connections per server
- **Host CPU contention** — TLS/OAuth2 processing steals cycles from MCP-serving hosts



DPUs are widely deployed across the data center

DPU one platform · many jobs running at once

Infrastructure Offload

via Red Hat DPU Operator + Cilium CNI Offload

Full CNI / eBPF networking on the DPU

Pod networking → L7 policy; host CPU freed

Managed by the DPU Operator

MCP Offload

via ktls-proxy

Offloads security, authentication and Layer-7 load balancing

THIS TALK

AI Inference Gateway Offload

via CNI Offload

LLM-d inference gateway (Envoy, EPP) + full datapath on the DPU

Secured RDMA Offload

via DRA NetSec

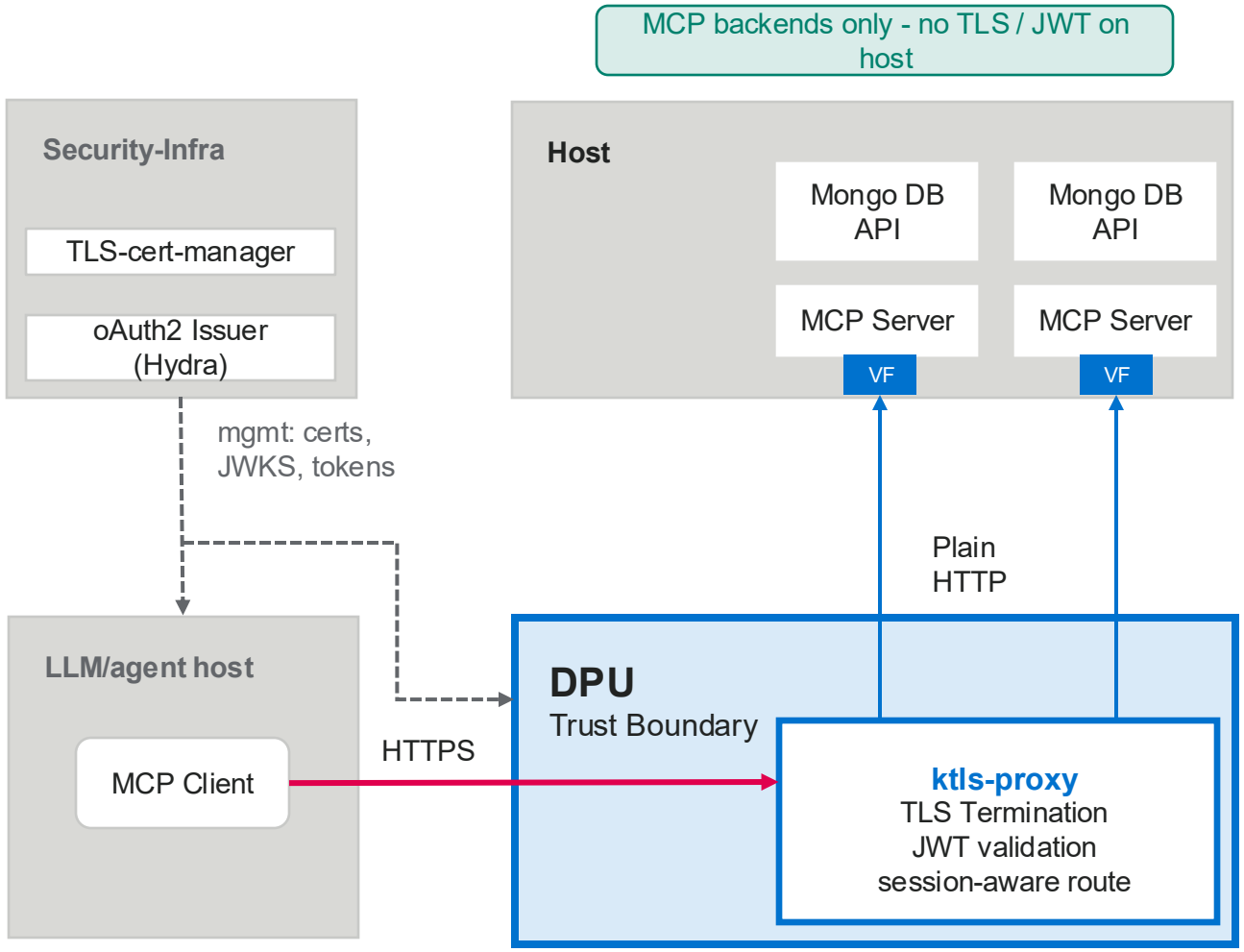
Offloads RDMA to the DPU + inline hardware MACsec encryption

→ made schedulable to Pods via DRA

Already on every node, no new hardware - MCP is just one more offload

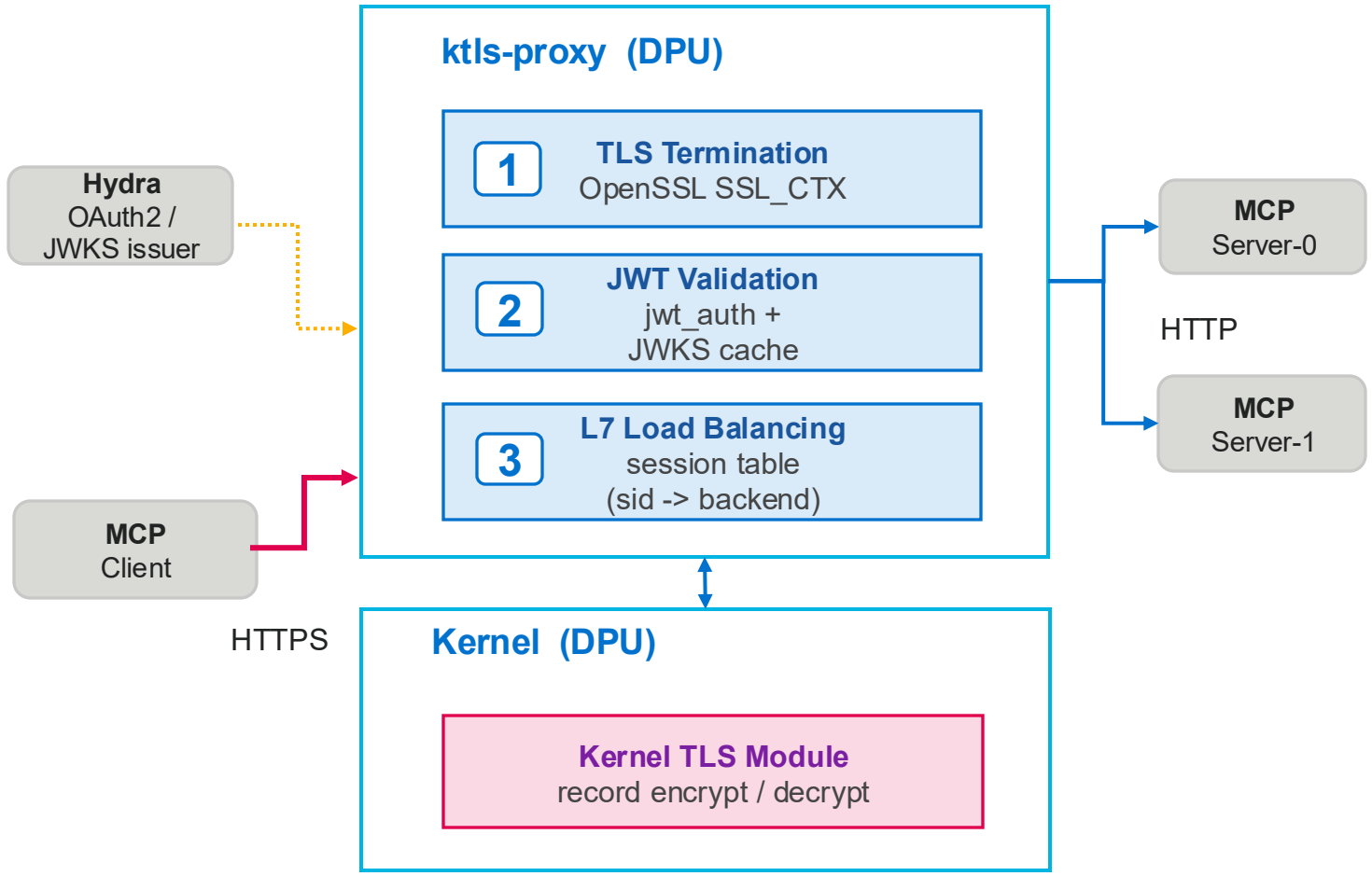
The DPU acts as the trust boundary for MCP traffic

- Offload security and traffic management to the DPU
- **Host isolation:** inference hosts run application logic only
- Security processing is isolated at the network edge
- Clients connect over HTTPS; MCP backends receive plain HTTP on an isolated network
- Invalid requests are rejected at the DPU — backends are never touched



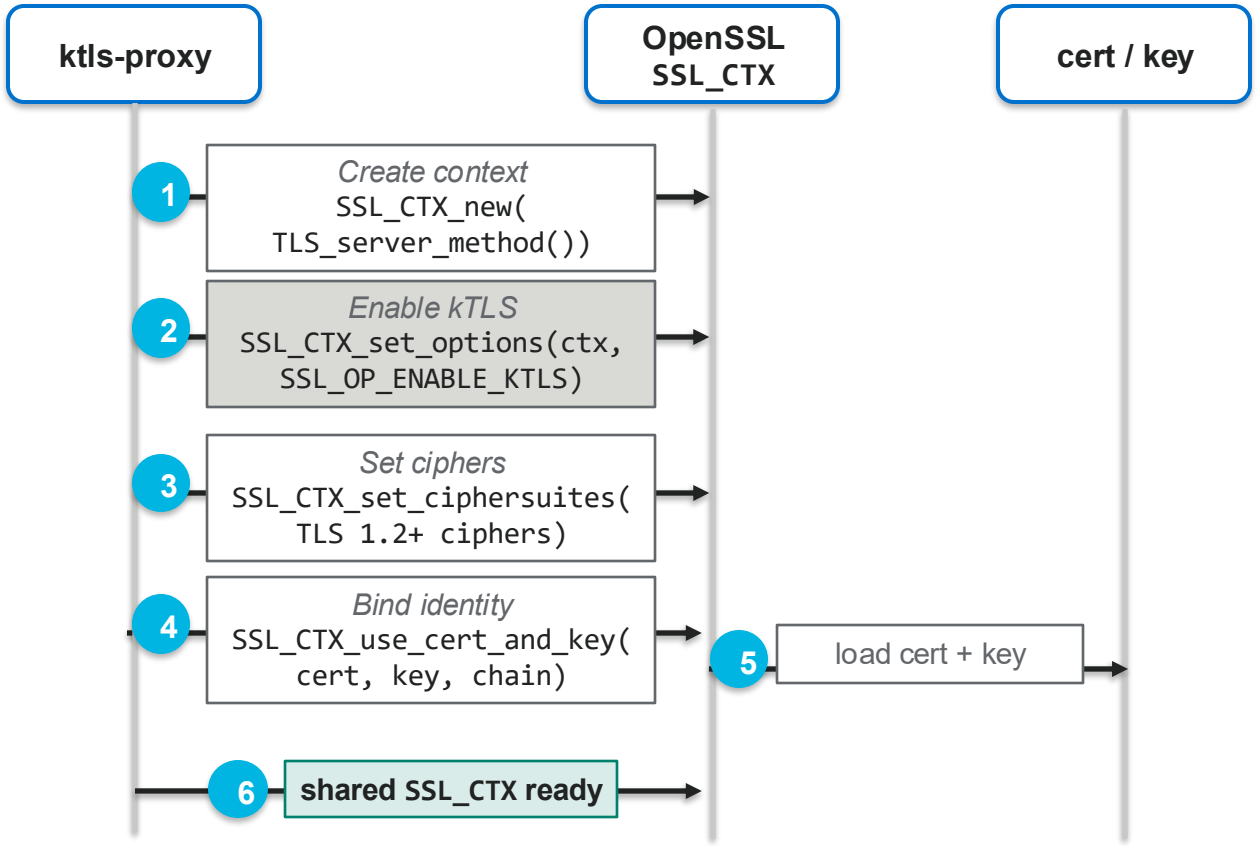
ktls-proxy architecture

- kTLS-proxy implements security and authentication
- Uses Kernel TLS module for record processing
- Handles the TLS handshake via OpenSSL
- Offloads TLS record processing to the kernel TLS (kTLS) module
- In-built JWT token validation against JWKS keys from issuer
- L7 load-balancing via in-process session table



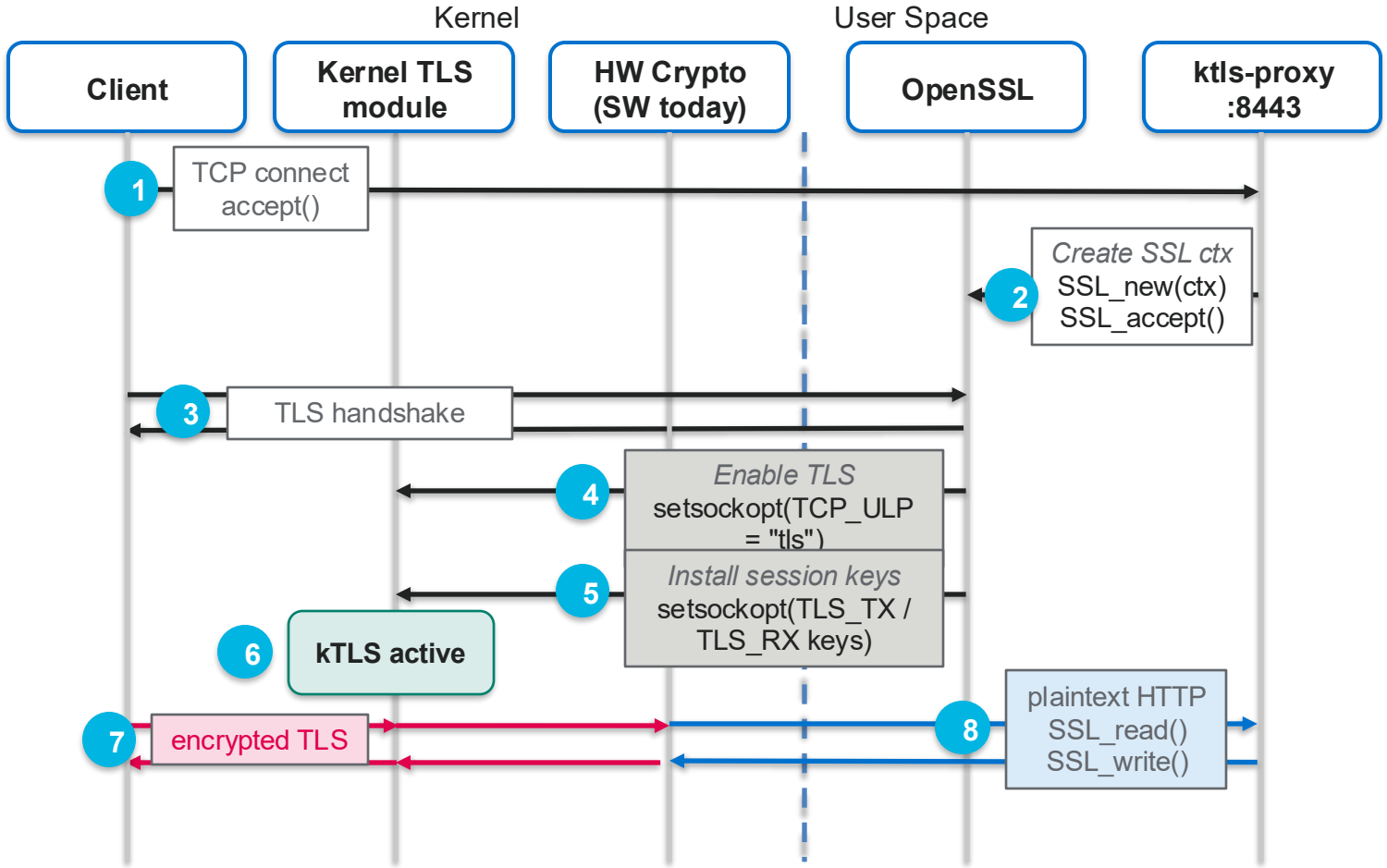
Enabling kTLS Using a Shared SSL_CTX

- Proxy create a shared **SSL_CTX** during startup
- Enable kernel TLS using **SSL_OP_ENABLE_KTLS**
- Configure supported TLS cipher suites
- Load server certificate and private key once
- All client connections reuse the shared **SSL_CTX**
- Every new TLS connection inherits the same kTLS-enabled configuration



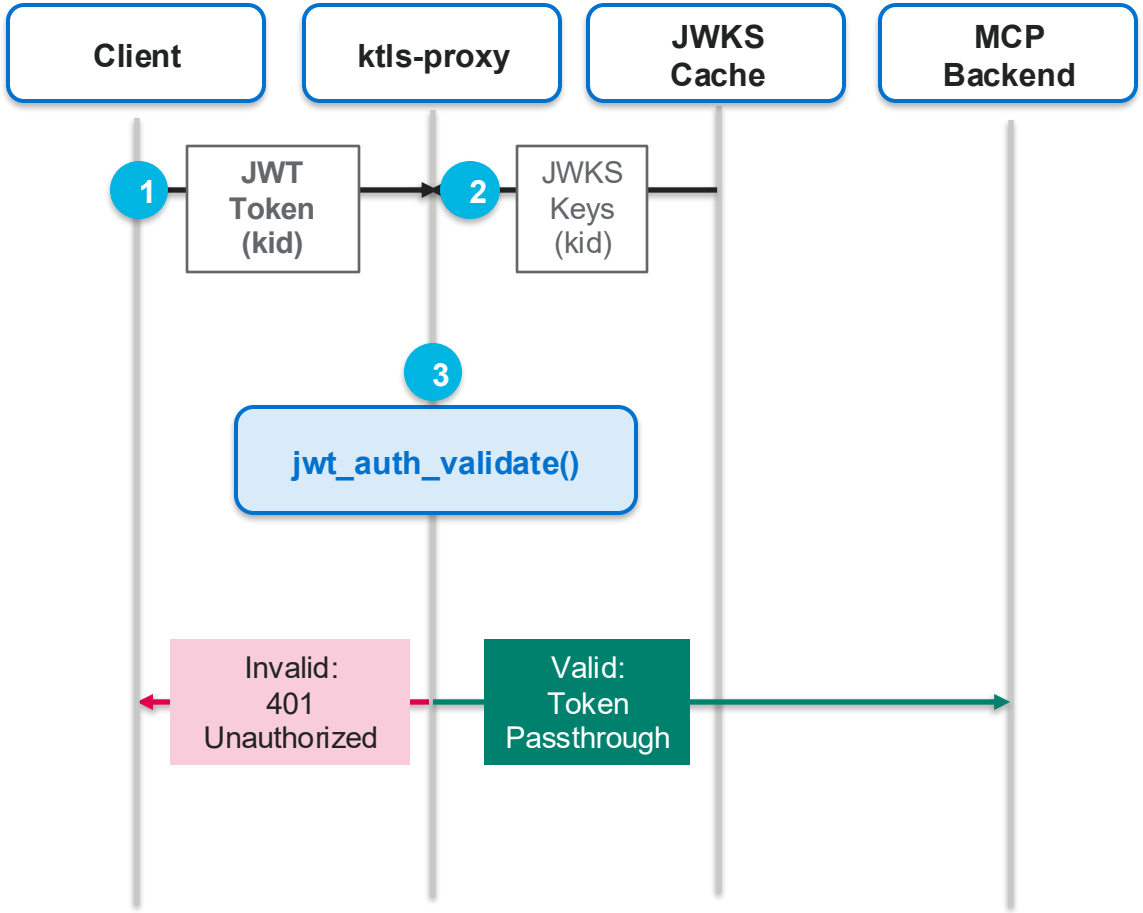
Each TLS connection hands off record processing to the kernel

- Accept a new TCP connection
- Create a per-connection **SSL** object from the shared **SSL_CTX**
- Complete the TLS handshake in OpenSSL (userspace)
- Install TLS session keys into the kernel (TCP_ULP=tls)
- TLS record encryption/decryption is offloaded to kTLS
- Proxy continue using **SSL_read()** / **SSL_write()**
- OpenSSL transparently hands TLS record processing to the kernel



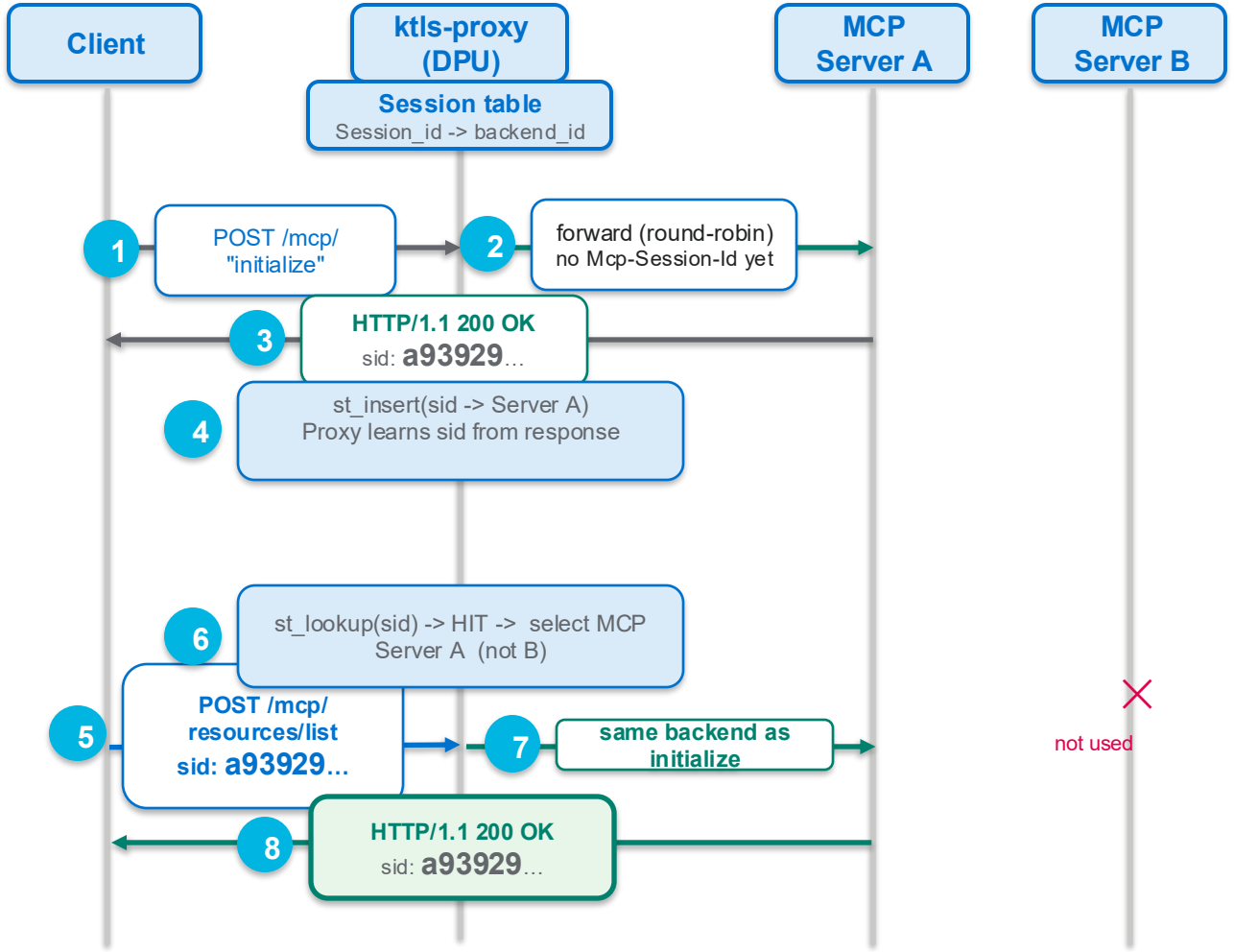
The DPU validates JWTs at the edge, before any backend

- Client sends `Authorization: Bearer <jwt>` inside HTTP Request
- Proxy verifies locally against **cached JWKS** - no per-request issuer call
- Invalid → `401 Unauthorized` at edge; request rejected before reaching any backend
- Valid → **token passthrough** - `Authorization` header forwarded unchanged
- Only authenticated requests reach MCP backends



Mcp-Session-Id enables session affinity

- MCP is **stateful** - after initialize, follow-up calls must reach the same backend
- Client: `POST/mcp {"method": "initialize"}` no session header yet
- Proxy picks backend via **round-robin** -> forwards to MCP Server A
- Response: `Mcp-Session-Id: a93929…` Proxy stores the mapping - `st_insert (sid -> Server A)` - **session CREATED**
- Client uses `Mcp-Session-Id: a93929…` in subsequent requests Proxy: `st_lookup (sid)` -> **session FOUND** -> same backend as initialize
- This session table allows the DPU to provide **Layer-7 affinity** independently of traditional load balancers



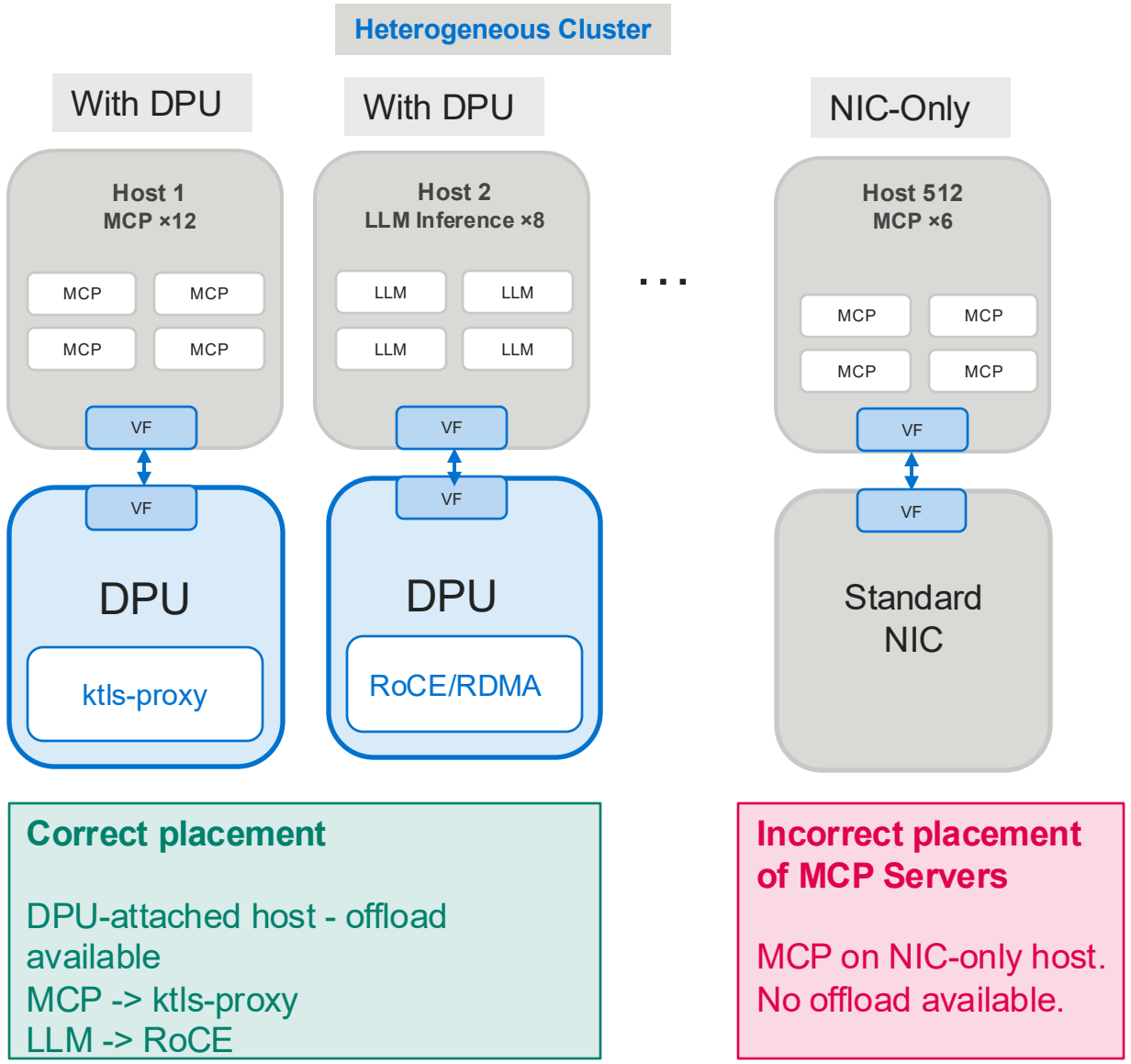
Operational challenges of deploying DPU offload at scale

Deployment
- Correct placement of ktls-proxy and data-plane configuration
- Co-location of ktls-proxy with MCP backends

Operations
- Manual VF provisioning
- Manual OVS datapath programming
- Configuration drift across DPUs

Cluster Management
- Mixed fleet of DPU-enabled and standard hosts
- Lifecycle management (upgrade, recovery)

Motivation
- **Kubernetes-based orchestration is required**



Bringing it together on Kubernetes

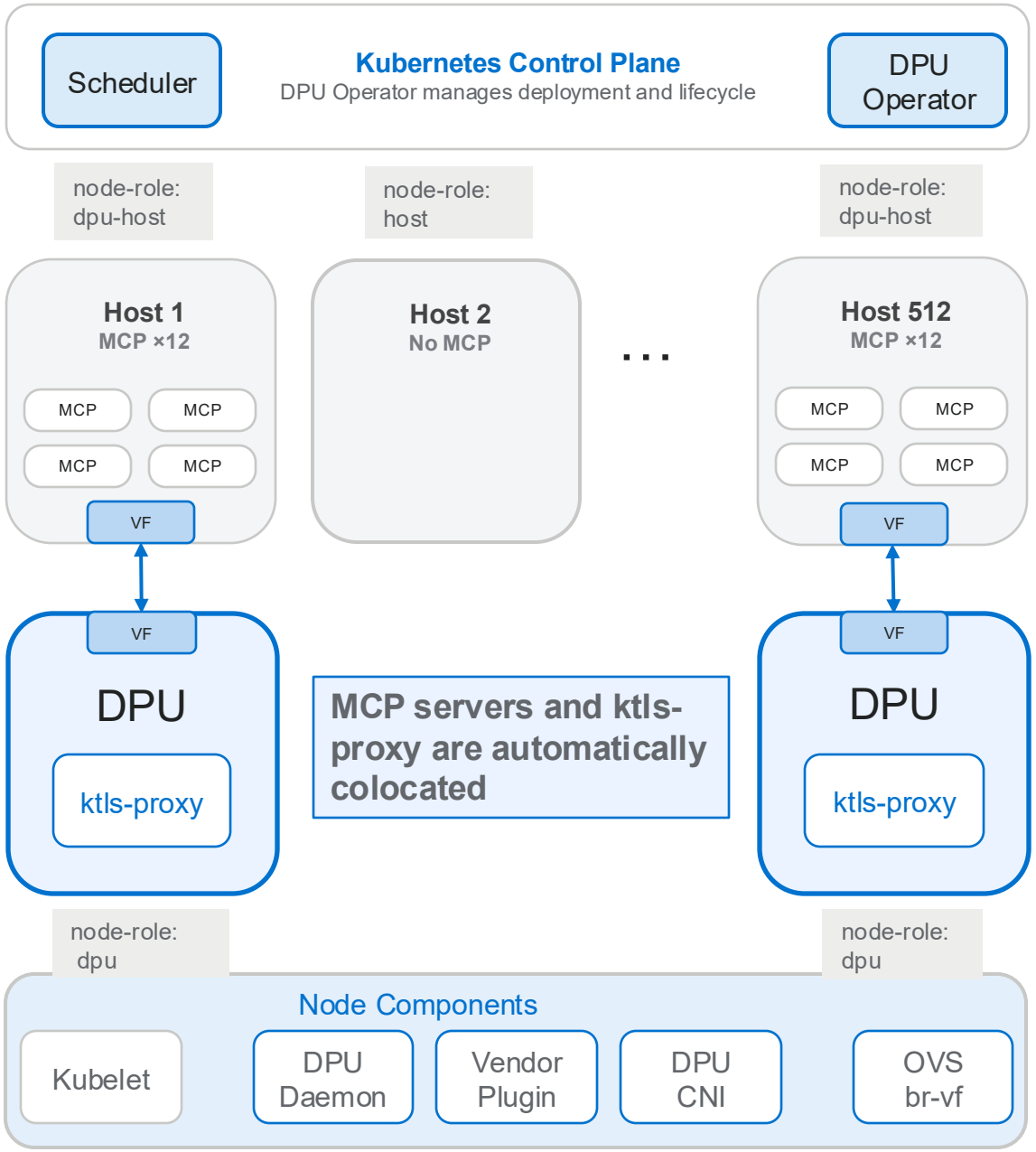
- **MCP** standardizes agent -> tool access - but puts TLS + JWT auth on every host
- **Kubernetes** automates discovery and placement
- **DPU Operator** enables VF provisioning and OVS networking
- **ktls-proxy** on the DPU offloads security and authentication
- **Result:** deploy secure MCP by manifest - no per-host TLS, VF, or datapath wiring

MCP backend

```
nodeSelector: dpu-host
networks: default-sriov-net
mongo-mcp :8080
openshift.io/dpu: 1
```

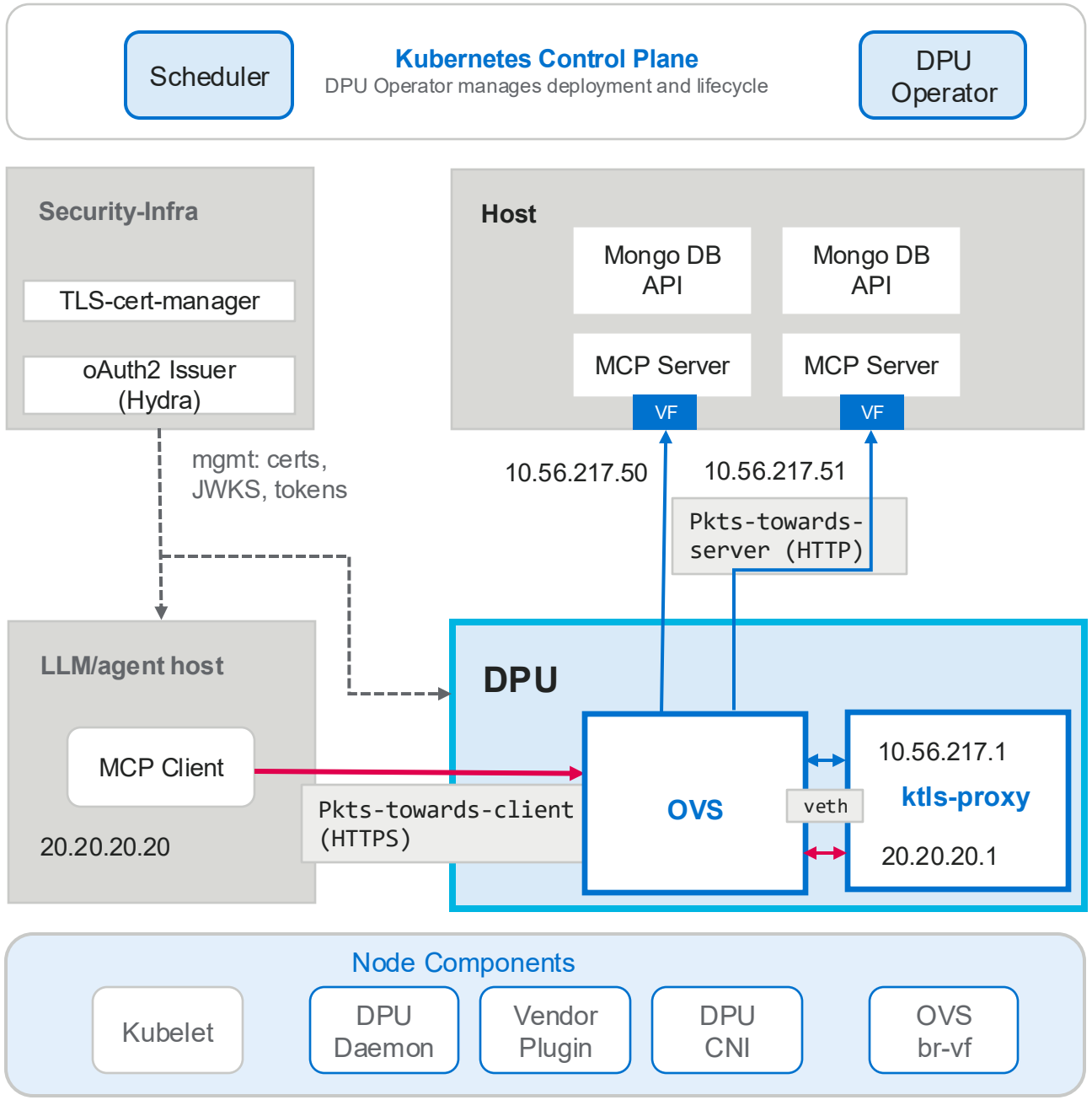
ktls-proxy NF

```
nodeSelector: dpu
networks: dpunfcni-conf ×2
ktls-proxy :8443
openshift.io/dpu: 2
```



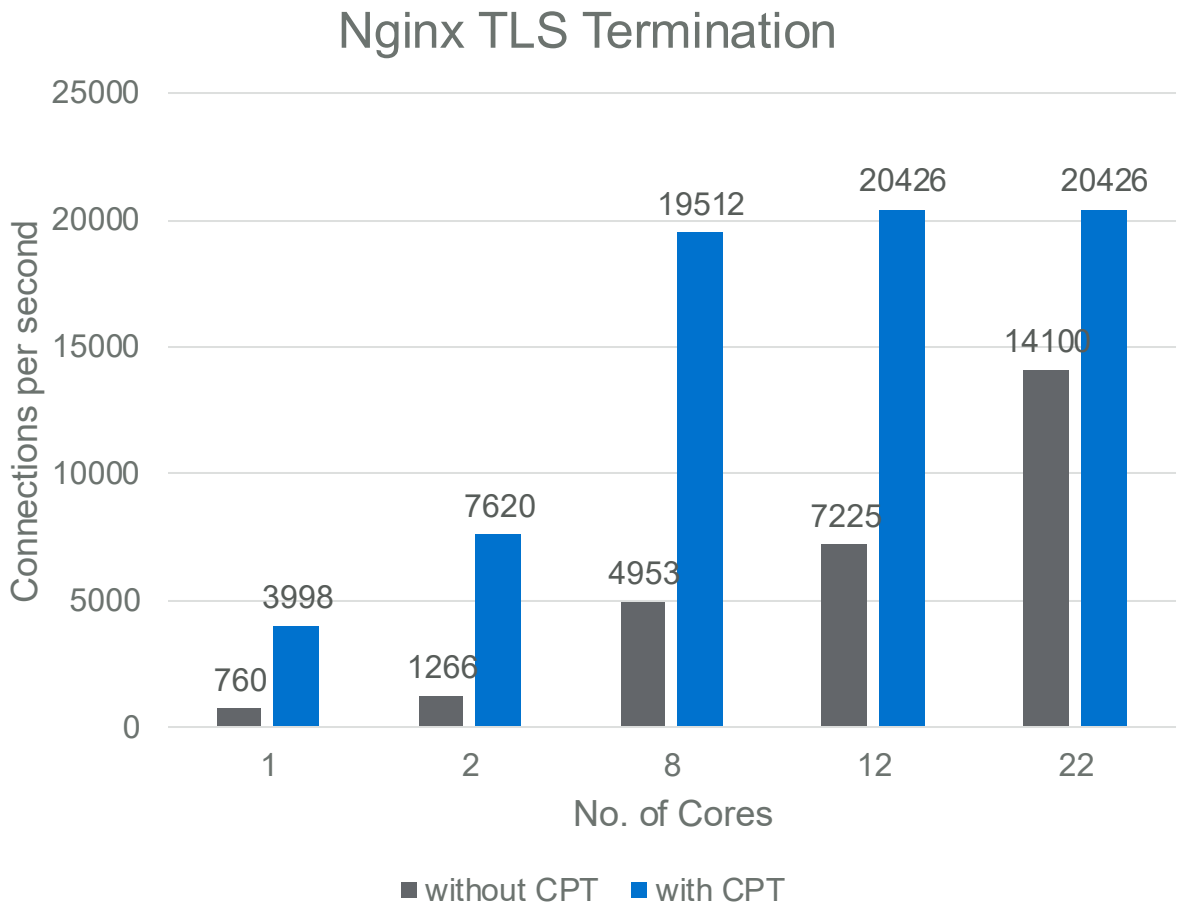
ktls-proxy lab setup

- Kubernetes cluster with Host + Marvell DPU
- Two MCP servers deployed on the host
- MCP client communicates over HTTPS
- DPU-resident ktls-proxy offloads TLS termination
- OVS forwards decrypted traffic to backend MCP servers
- Certificates and OAuth2 credentials managed by the cluster



Performance impact of crypto HW offload

- Crypto HW (CPT) offload delivers up to 5.3× TLS throughput gain
  - 1 core: ≈ 5.3× performance gain
  - 12 cores: ≈ 2.8× performance gain
- TLS Version: 1.2
- Cipher: AES256-GCM-SHA384
- DPU: CN106x
- Clock Values:
  - RCLK: 2500 MHz
  - SCLK: 1100 MHz
  - MESHCLK: 1900 MHz



Future work - automated backend lifecycle

- Today: static backend = lines in ktls-proxy.conf;
  Next: operator-driven backend discovery and reconcile
- **Discover** MCP backends per host
- **Publish** desired backends to a ConfigMap
- **Reload** ktls-proxy when the ConfigMap changes
- **Reconcile** OVS datapath (br-mrv0, VF reps) on every pod add/remove
- **Evict** stale session-table entries when backends scale in
- Operator discovers MCP backends and keeps ktls-proxy in sync

Thank You