
# Nerpa: Network Programming with Relational and Procedural Abstractions

*Debnil Sur*, Ben Pfaff, Leonid Ryzhyk, Mihai Budiu

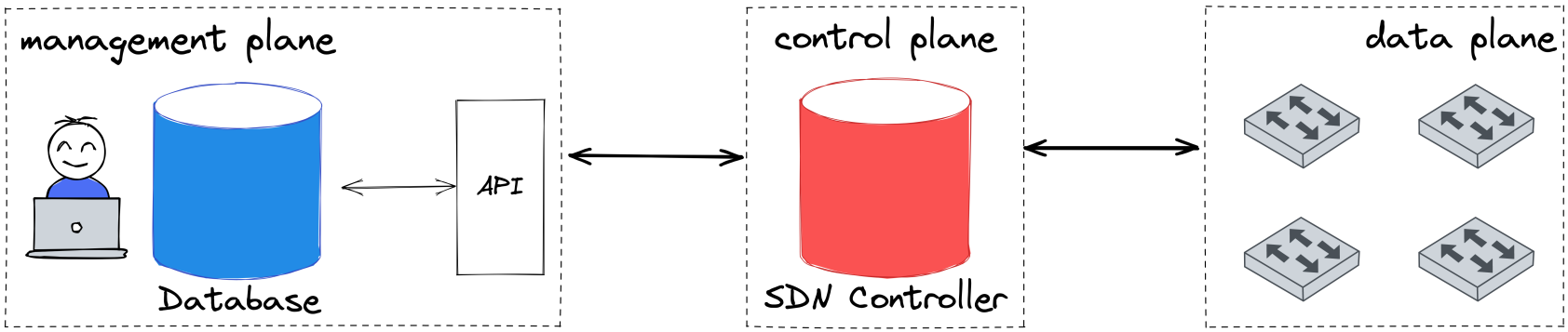
VMware

A unified environment for network programming.

# Roadmap

- Motivation
- Design
- Example
- Questions

# Motivation

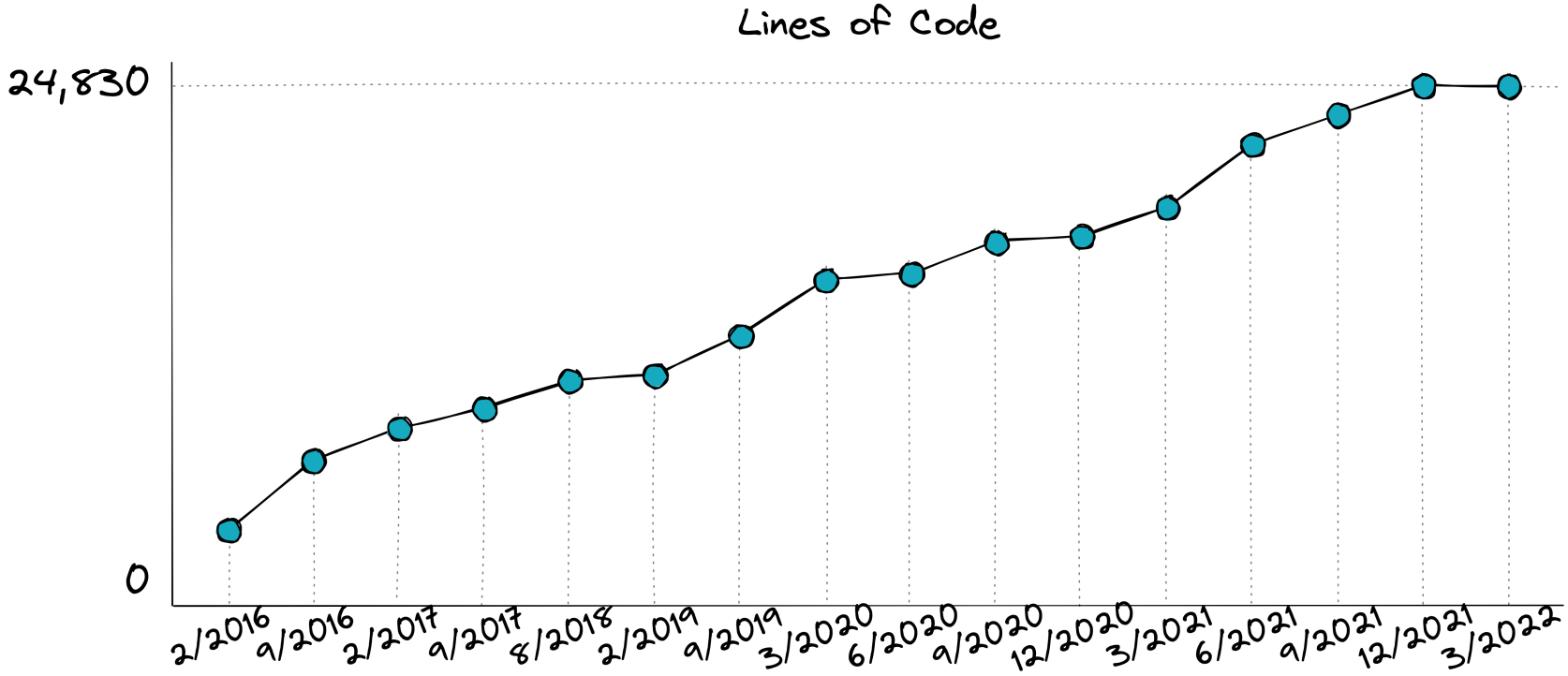



Management, control, and data planes are usually programmed *separately*.

# Problem #1: Correctness

- Difficult to maintain SDN systems over time
- High-level policies $\rightarrow$ program fragments $\rightarrow$ network devices
- Flow rule fragments become scattered in a growing codebase

# Problem #1: Correctness

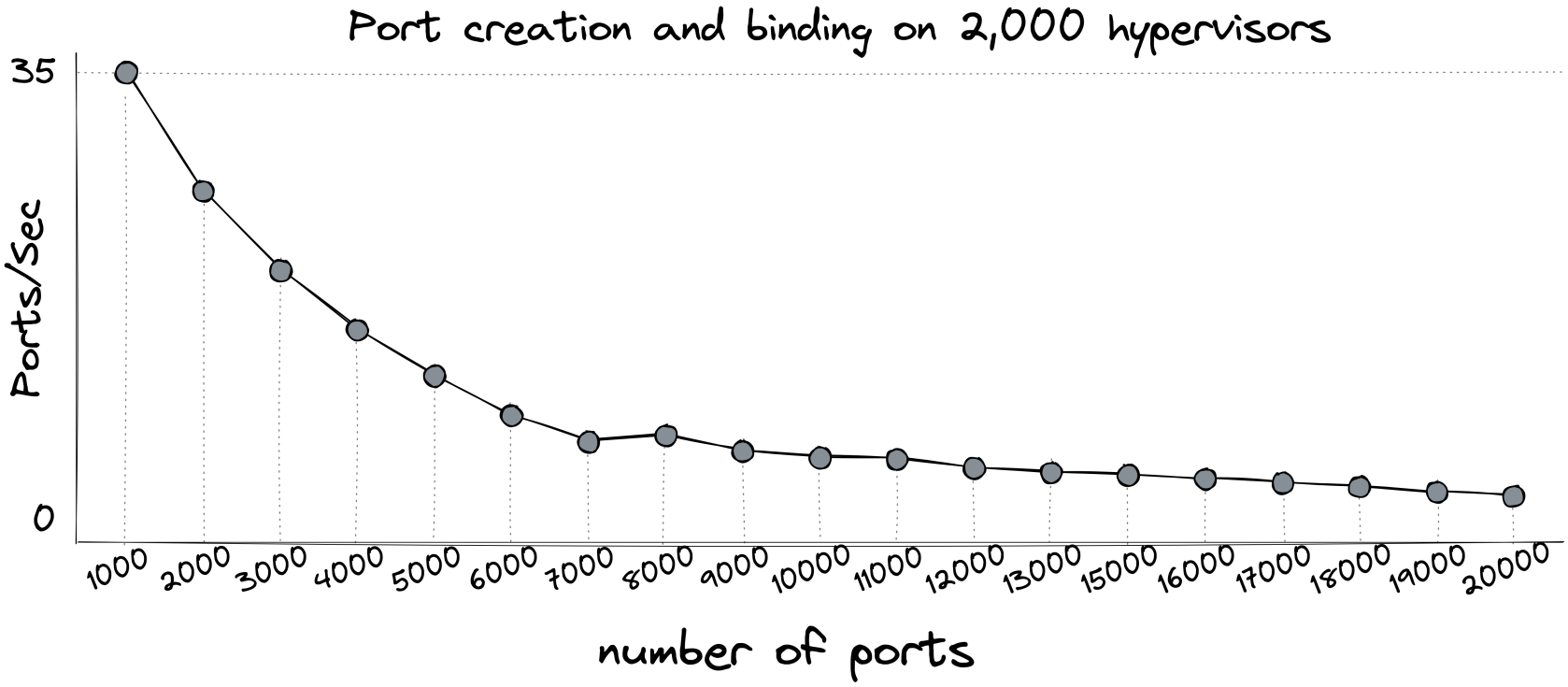



Size of `northd` in OVN: 2,608 LOC in 2016 to 24,830 in 2021

# Problem #2: Scalability

- Demands incrementality
- Incremental programming is hard, ugly, and error-prone!

# Problem #2: Scalability




- Scale of OVN in 2016
- Context: in L2 mode, with independent and local logical switches

# Nerpa

- Prototype of programming framework
- Automatically incremental control plane in Differential Datalog (DDlog) allows scalability
- Co-designed control and data planes makes data conversion easier

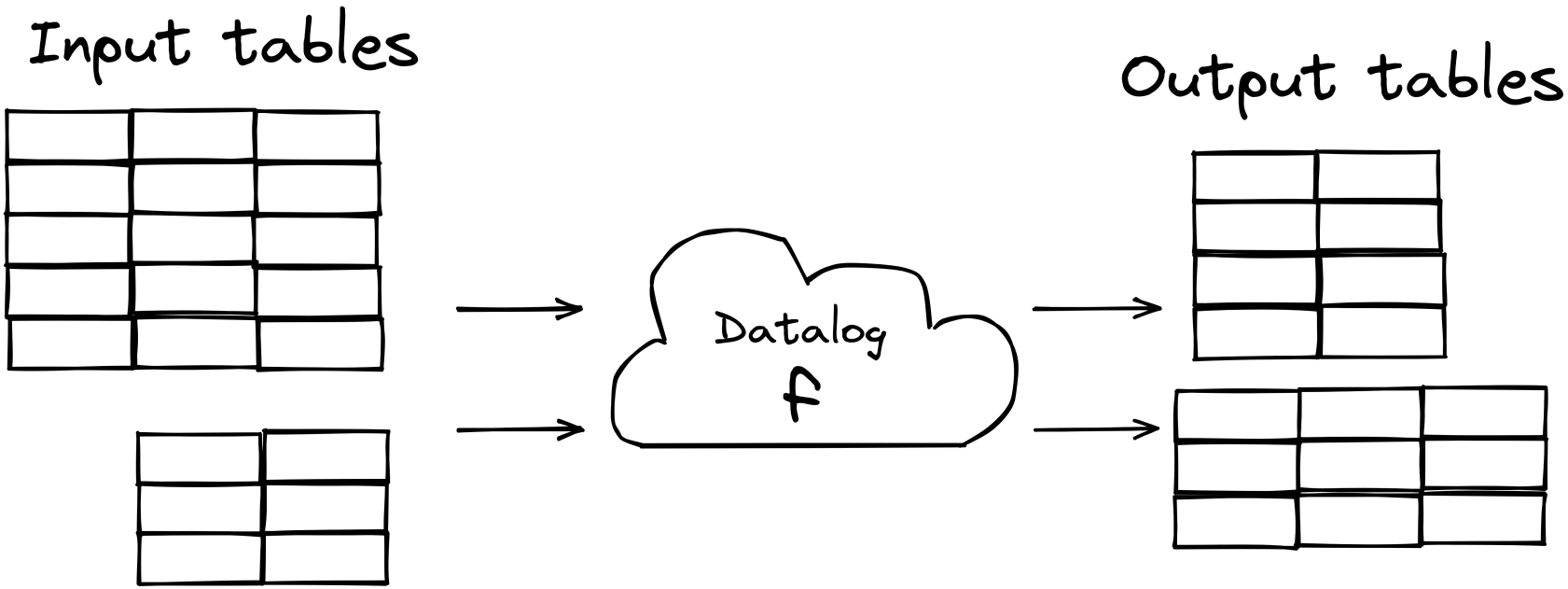
# Solution: Correctness

DDlog control plane + P4 data plane = type-checked program

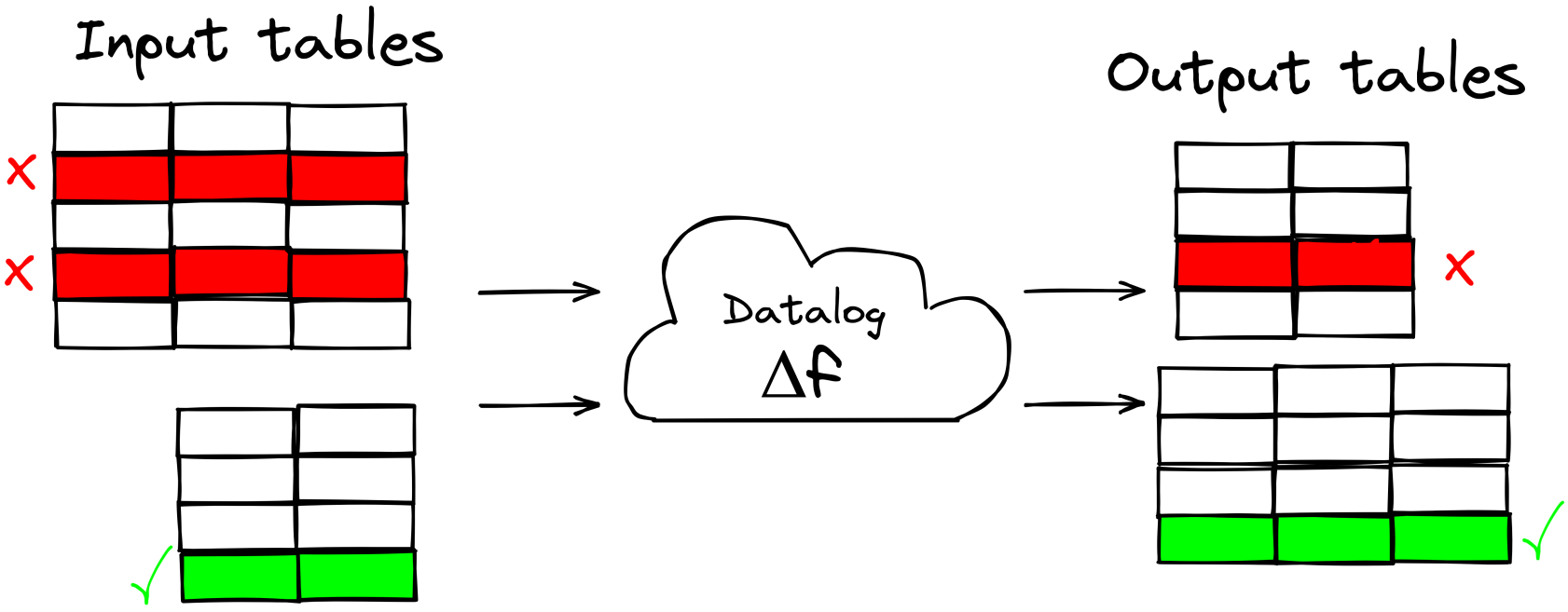
# Solution: Scalability

- DDlog control plane **is** incremental!
- More benefits: relational, dataflow-oriented, typed, and more

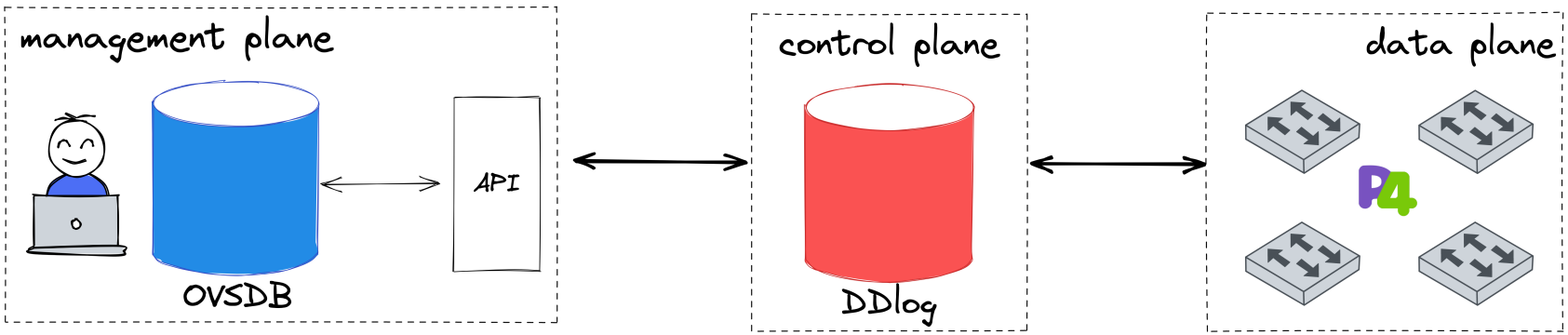
# Solution: Scalability



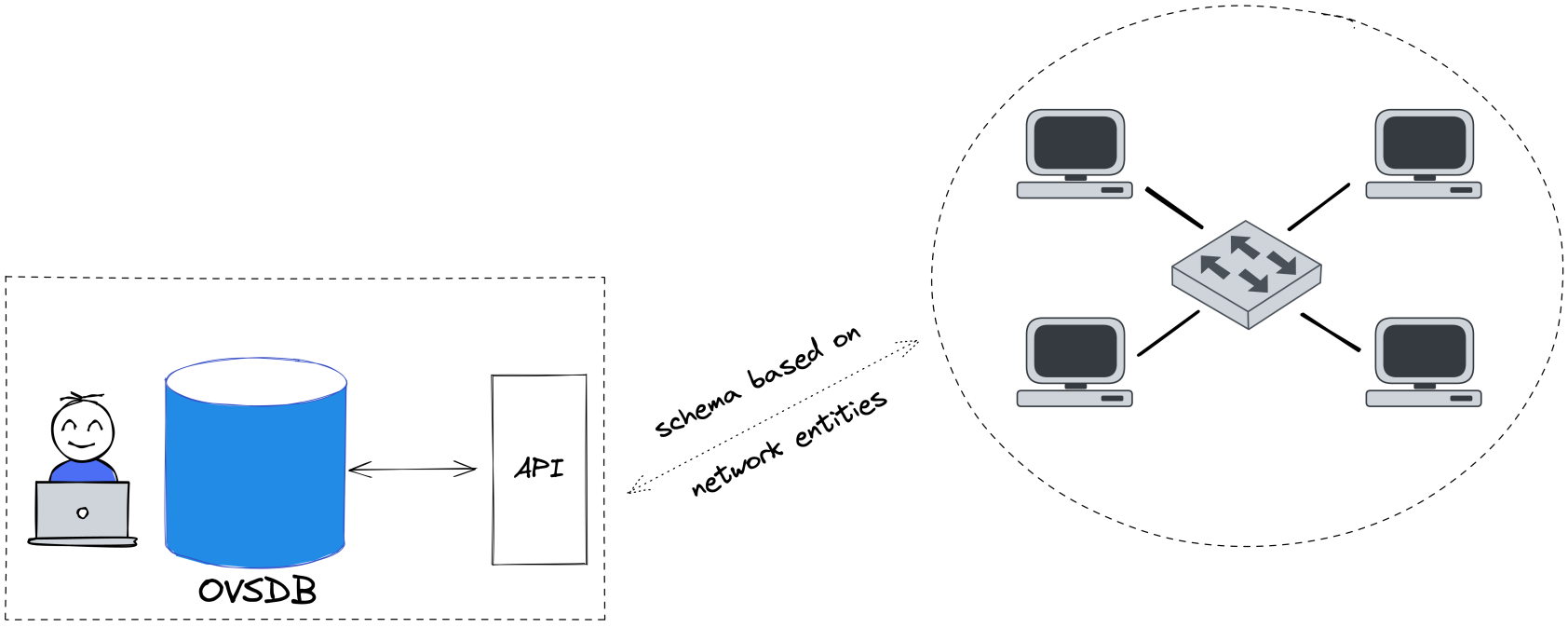
# Solution: Scalability



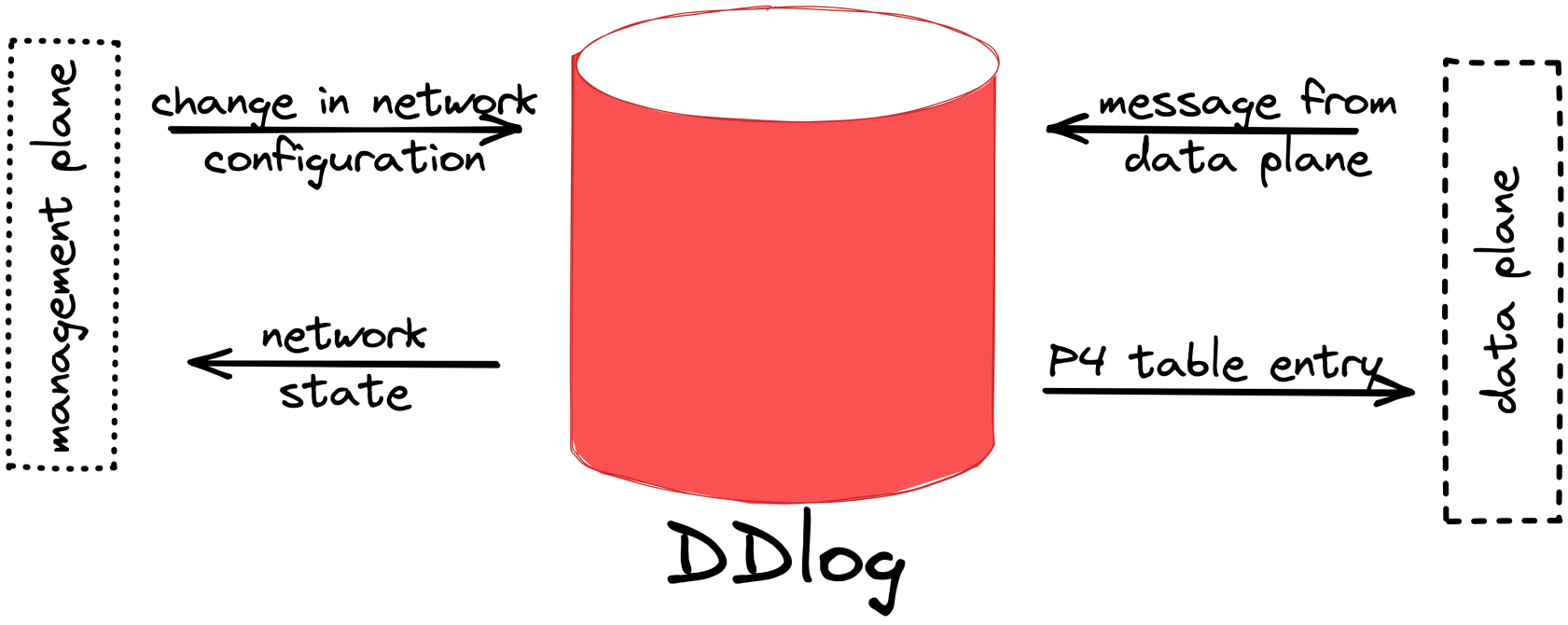
# Design


management plane
API
OVSDB
control plane
DDlog
data plane
P4

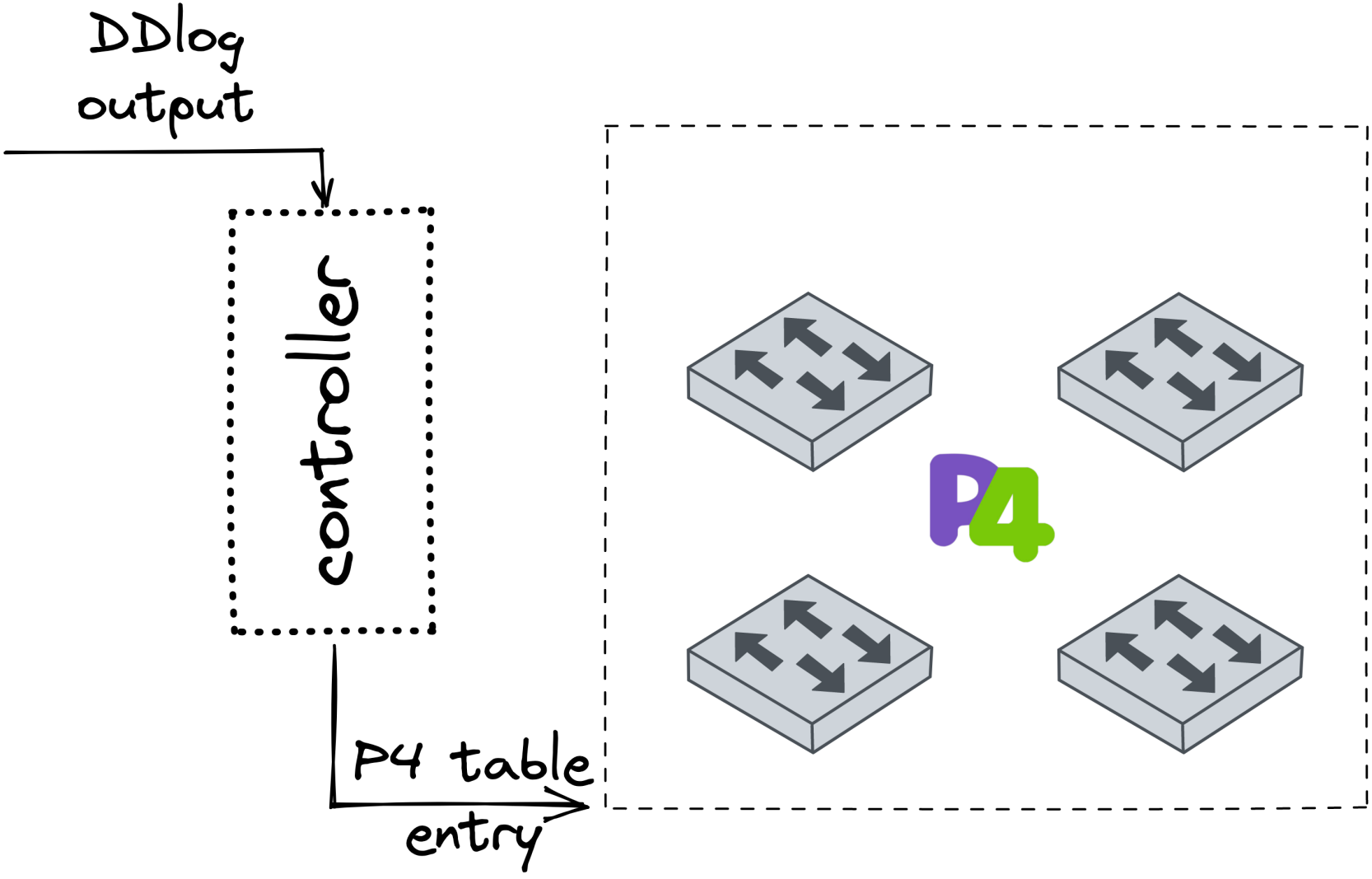
# Design: Management plane



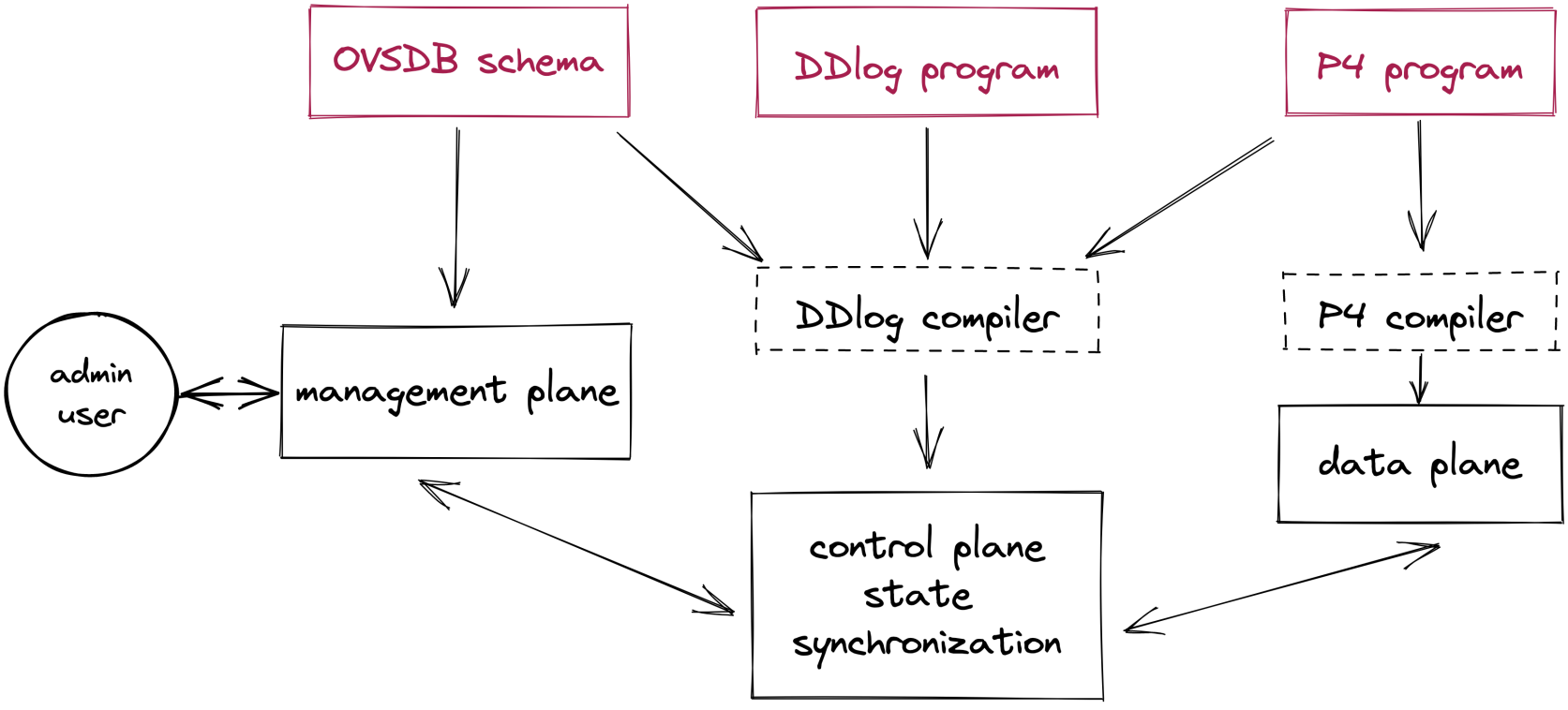
# Design: Control plane



# Design: Data plane

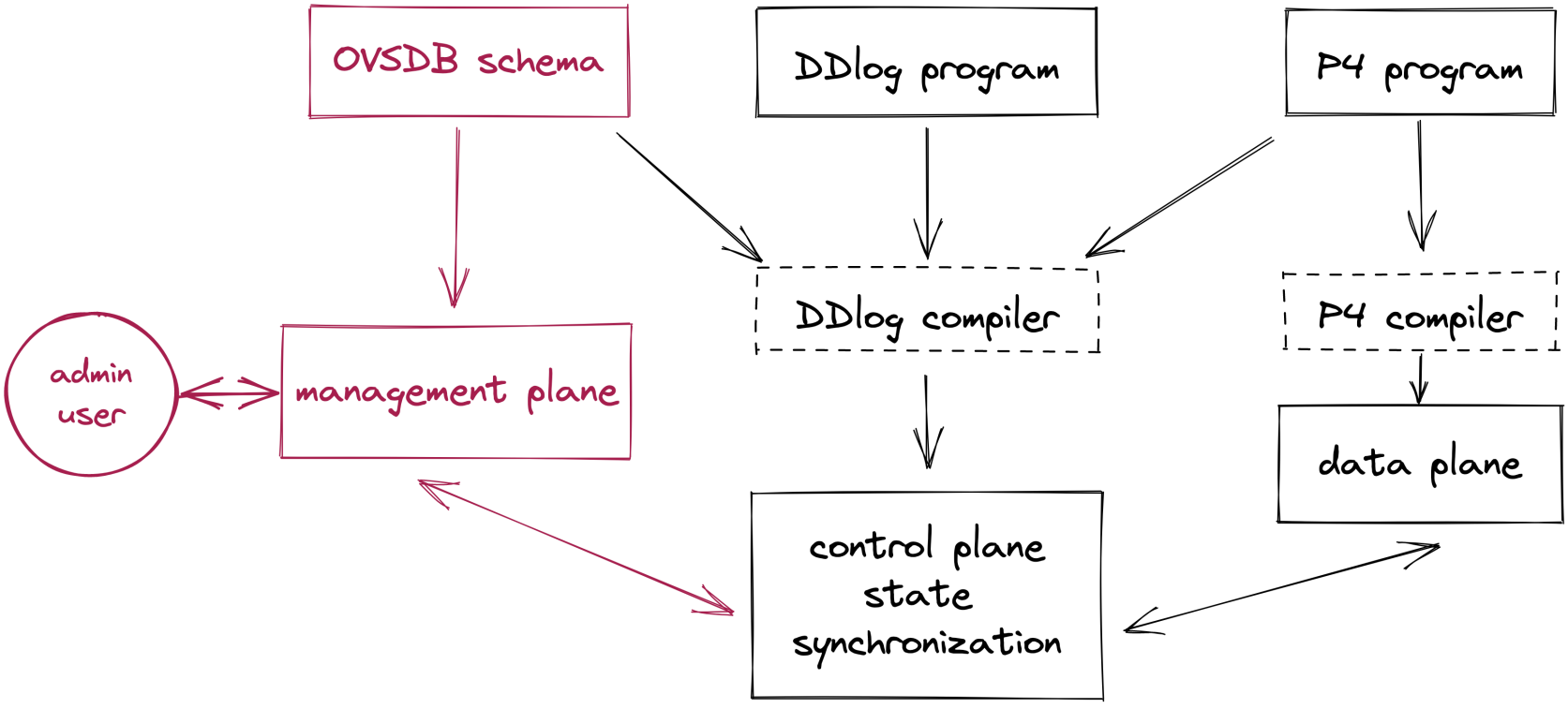


# Putting it all together...



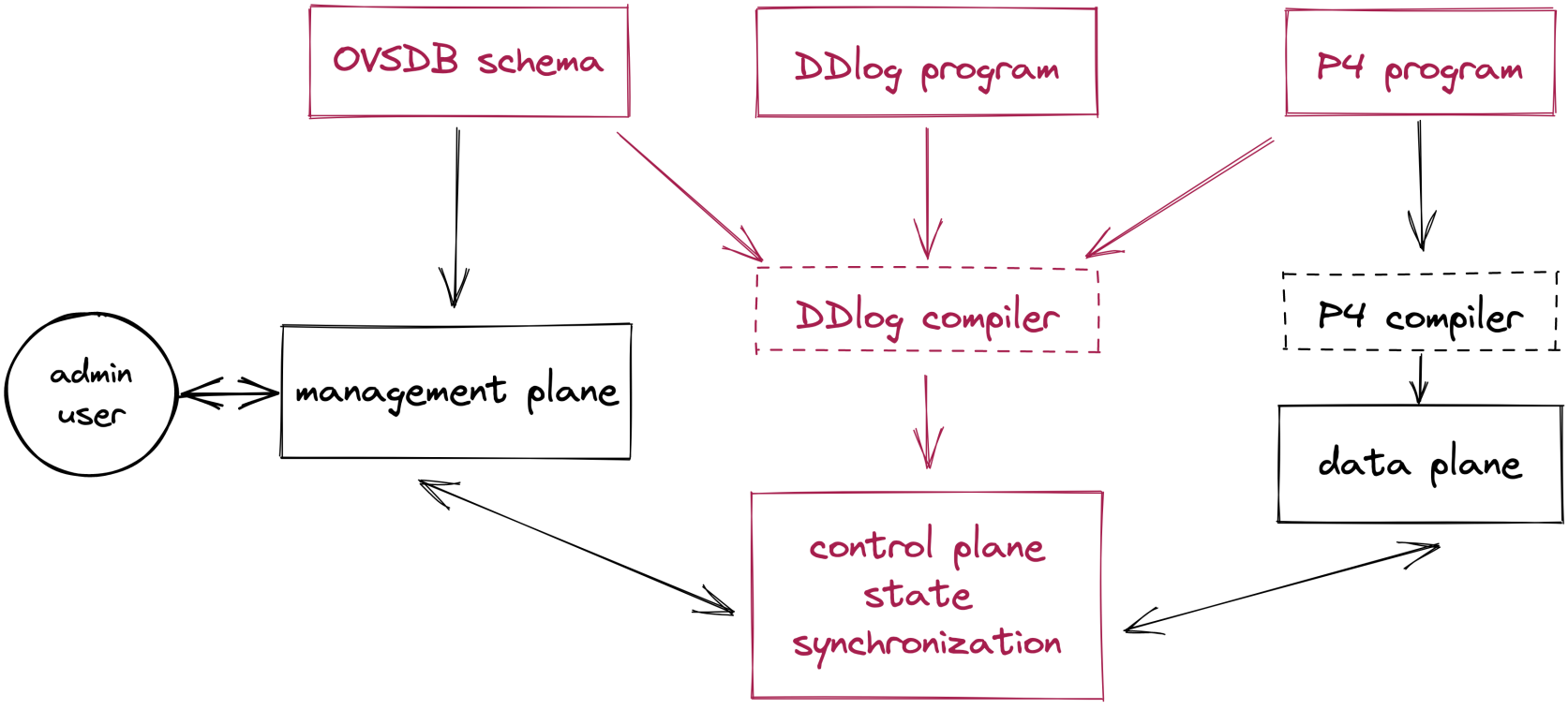

- How information flows in Nerpa
- Red boxes are user-provided files

# Putting it all together...



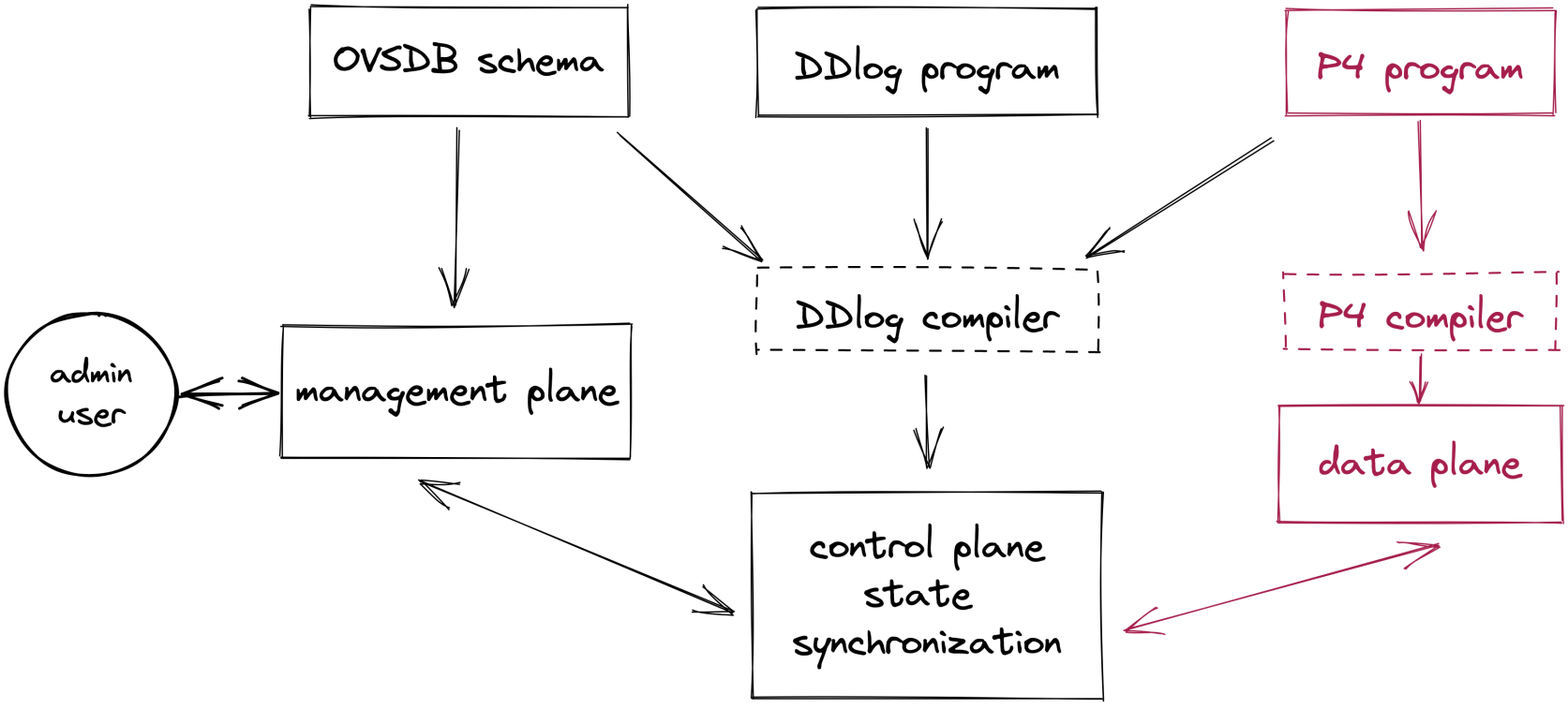

Management plane

# Putting it all together...




Control plane and state synchronization

# Putting it all together...




Data plane

# Example: Simple Network Virtual Switch

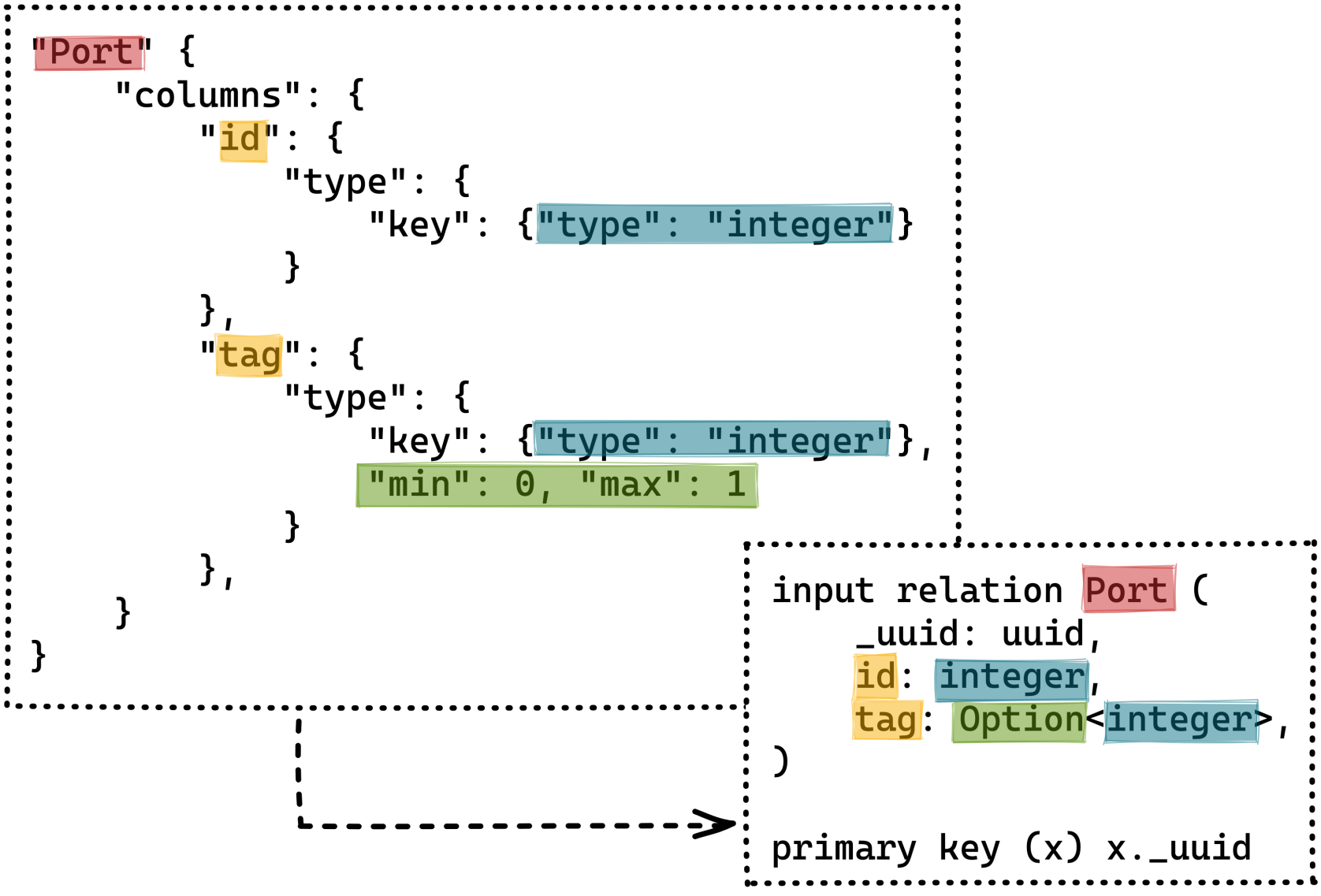
- VLANs, MAC learning, port mirroring
- Used in Nerpa integration test

# P4 to DDlog output

```
table InVlan {
    key = {
        std_meta.in_port: exact @name("port");
        hdr.vlan.isValid(): exact @name("has_vlan") @nerpa_bool;
        hdr.vlan.vid: optional @name("vid");
    }

    actions = {
        Drop;
        SetVlan;
        UseTaggedVlan;
    }
}
```

```
output relation InVlan(
    port: bit<9>,
    has_vlan: bool,
    vid: Option<bit<12>>,
    priority: bit<32>,
    action: InVlanAction
)

typedef InVlanAction = InVlanActionDrop
    | InVlanActionSetVlan {vid: bit<12>}
    | InVlanActionUseTaggedVlan
```

# OVSDB to DDlog input



# DDlog rule

```
input relation Port (
    _uuid: uuid,
    id: integer,
    tag: Option<integer>,
)

primary key (x) x._uuid
```

```
output relation InVlan (
    port: bit<9>,
    has_vlan: bool,
    vid: Option<bit<12>>,
    priority: bit<32>,
    action: InVlanAction
)
```

```
InVlan(port, false, None, 1, InVlanActionSetVlan{vid}) :-
    Port(.id = port, .tag = tag),
    var vid = match tag {
        None -> 0,
        Some{v} -> v
    }.
```

# Conclusion

- Relational and procedural abstractions can improve correctness and scalability
- github.com/vmware/nerpa, with tutorial, demo, and useful libraries
- Future: Implement more things!


# Thank You

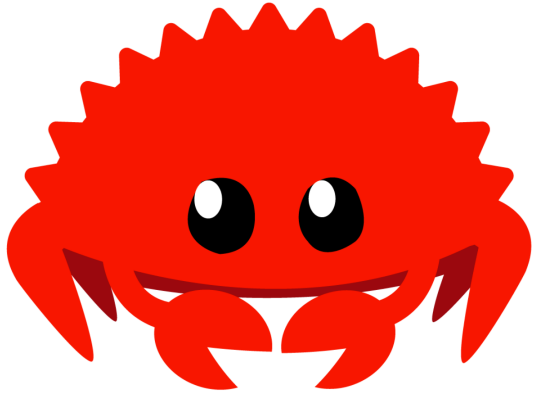
github.com/vmware/nerpa

# Solution: Scalability

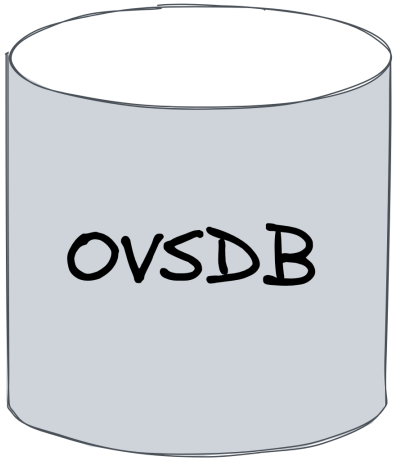
```
input relation Edge(from: Node, to: Node)
output relation Reachable(from: Node, to: Node)

Reachable(from, to) :- Edge(from, to).
Reachable(from, to) :- Reachable(from, via), Edge(via, to).
```

- Network reachability in four lines of DDlog
- Equivalent Java version is several thousand LOC

# Implementation: Language Tooling

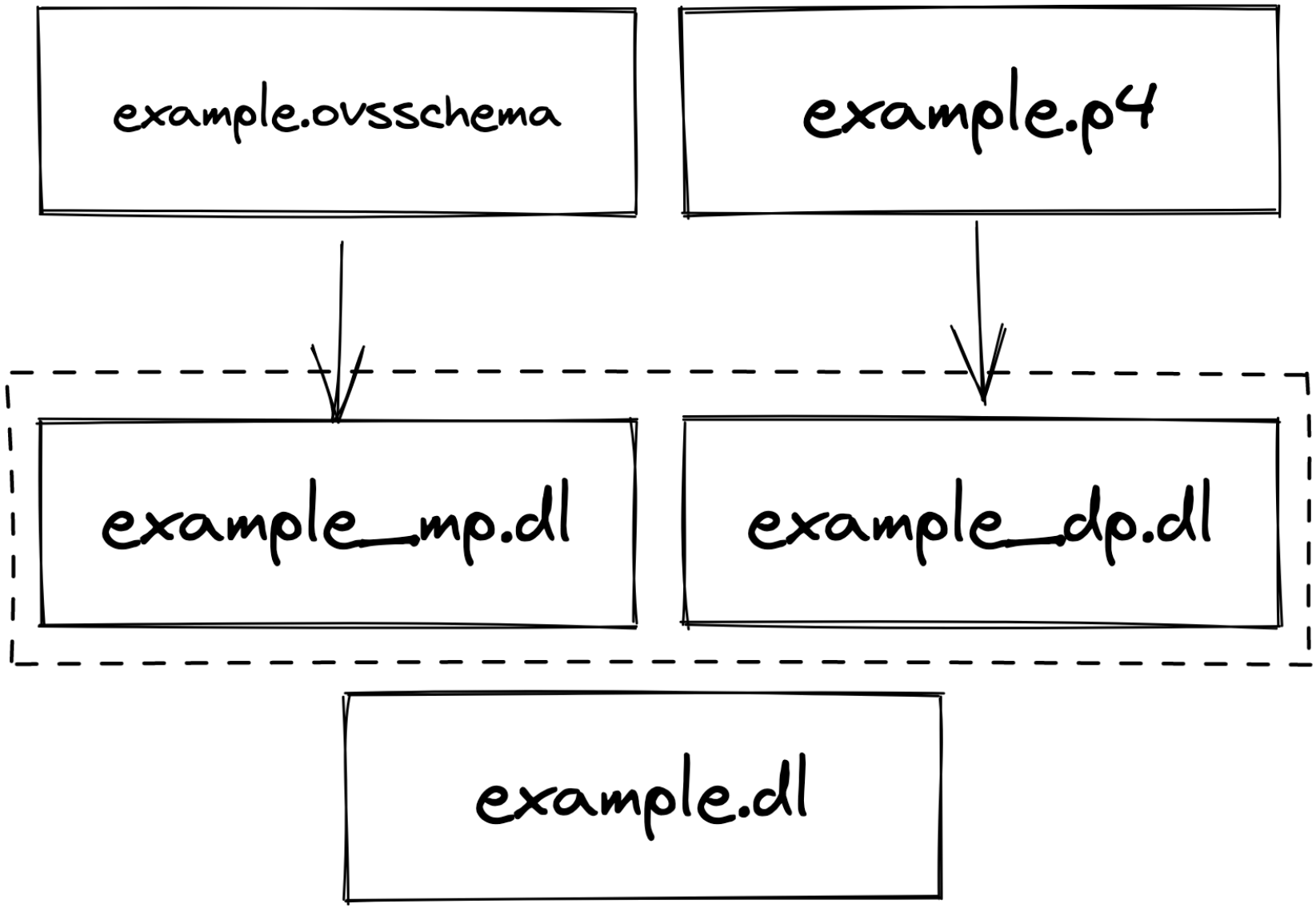


Rust




Runtime

# Implementation: Control/Data Plane Co-Design




Control plane relations are generated from the data and management planes